

# On Proving Confluence Modulo Equivalence for Constraint Handling Rules

Henning Christiansen<sup>1</sup> and Maja H. Kirkeby<sup>1,2</sup>

Research group PLIS: Programming, Logic and Intelligent Systems  
Department of People and Technology  
Roskilde University, Denmark

**Abstract.** Previous results on proving confluence for Constraint Handling Rules are extended in two ways in order to allow a larger and more realistic class of CHR programs to be considered confluent. Firstly, we introduce the relaxed notion of confluence modulo equivalence into the context of CHR: while confluence for a terminating program means that all alternative derivations for a query lead to the exact same final state, confluence modulo equivalence only requires the final states to be equivalent with respect to an equivalence relation tailored for the given program. Secondly, we allow non-logical built-in predicates such as `var/1` and incomplete ones such as `is/2`, that are ignored in previous work on confluence.

To this end, a new operational semantics for CHR is developed which includes such predicates. In addition, this semantics differs from earlier approaches by its simplicity without loss of generality, and it may also be recommended for future studies of CHR.

For the purely logical subset of CHR, proofs can be expressed in first-order logic, that we show is not sufficient in the present case. We have introduced a formal meta-language that allows reasoning about abstract states and derivations with meta-level restrictions that reflect the non-logical and incomplete predicates. This language represents subproofs as diagrams, which facilitates a systematic enumeration of proof cases, pointing forward to a mechanical support for such proofs.

## 1. Introduction

Constraint Handling Rules, CHR [21, 22], is a programming language consisting of guarded rewriting rules over constraint stores. CHR inherits its nomenclature from the logic programming tradition; constraints are first-order atoms, and the language has a declarative semantics based on a logical reading of the rules. It has become important as a general language for knowledge representation and reasoning as well as for expressing algorithms in a high-level fashion; see, e.g., [18, 37, 39].

A foundation for applying confluence in the analysis and verification of CHR programs has been laid in earlier work, and the overall theoretical issues are well understood [1, 3, 4, 18]. The confluence notion goes

---

<sup>1</sup> The project is supported by The Danish Council for Independent Research, Natural Sciences, grant no. DFF 4181-00442

<sup>2</sup> The second author's contribution received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318337, ENTRa - Whole-Systems Energy Transparency.

longer back in the traditions of term and abstract rewriting systems; see more details in Section 2.1. There are, however, still severe limitations in the results for CHR that impede its practical application to realistic programs. The present paper aims at filling part of the gap, by

- the introduction for CHR of confluence *modulo equivalence* that allows a much larger and interesting class of programs to enjoy the advantages of confluence;
- extending to a larger subset of CHR that includes non-logical and incomplete<sup>3</sup> built-in predicates (e.g., `var/1`, resp. `is/2`) that have been ignored in previous work.

While confluence of a program means that all derivations from a common initial state end in the same final state, the “modulo equivalence” version relaxes this such that final states need not be strictly identical, but only equivalent with respect to a given equivalence relation. The following motivating example is used throughout this paper.

**Example 1 ([12]).** The following CHR program, consisting of a single rule, collects a number of separate items into a (multi-) set represented as a list of items.

```
set(L), item(A) <=> set([A|L]).
```

This rule will apply repeatedly, replacing constraints matched by the left-hand side by those indicated to the right. The query

```
?- item(a), item(b), set([]).
```

may lead to two different final states,  $\{\text{set}([a, b])\}$  and  $\{\text{set}([b, a])\}$ , both representing the same set. We introduce a state equivalence relation  $\approx$  implying that  $\{\text{set}(L)\} \approx \{\text{set}(L')\}$ , whenever  $L$  is a permutation of  $L'$ . The program is not confluent when identical end states are required, but it will be shown to be confluent modulo  $\approx$  in Section 6.1 below.

The relevance of confluence modulo equivalence is also demonstrated for dynamic optimization programs that produce an arbitrary, optimal solution among a collection of equally good ones; the Viterbi algorithm expressed in CHR is considered in Section 6.2.

To model non-logical and incomplete predicates, we need to introduce a new operational semantics for CHR. To be interesting for studies of confluence, this semantics maintains nondeterminism for choice of the next rule to be applied to the current state. In addition to treating a larger language, this semantics differs from earlier approaches by its simplicity without loss of generality. Various redundancies have been removed so that a program state has only two components, a constraint store and a bookkeeping device to handle well-known termination problems for the propagation rules of CHR; a simple observation shows that global variables are unnecessary; execution of built-in predicates are modelled by substitutions applied to the state immediately, which is more in line with how a practical CHR system works (as opposed to earlier proposals’ additional store of “processed” built-ins and their evaluation explained by logical entailment). A detailed comparison and references to previous operational semantics are given in Section 3 below.

Reasoning about derivations is more difficult in the context of non-logical/incomplete built-ins. Basically, all earlier proof methods for the purely logical subset of CHR rely on a subsumption principle that any property shown about derivations between states also holds when more constraints are added and substitutions applied to the states; as a consequence of this, confluence proofs can be reduced to considering a finite number of cases that can be checked in an automatic way. This principle breaks down when non-logical predicates are introduced, e.g., the predicate `var(X)` succeeds but the instance `var(7)` fails. To cope with this, we have introduced a formal meta-language METACHR to represent abstract states, derivations and proofs as diagrams, with powerful parametrization and meta-level constraints that limit the allowed instances. The following is an example of an abstract term in the meta-language, `var(a) WHERE variable(a)`. Here,  $a$  is a meta-variable ranging over terms and *variable* is a meta-level constraint on such terms, allowing only substitutions to names of such variables. This abstract term is said to cover all instances that satisfy the meta-level constraint, i.e., `var(X)` but not `var(7)`. METACHR allows us to reason about such abstract terms in a way so that properties shown at this level are guaranteed to hold for all such permissible instances. We can demonstrate that proofs of confluence can be reduced to considering only a finite number of abstract

<sup>3</sup> In this paper, we use the term *incomplete* for a built-in predicate whose (established) implementation produces runtime errors for selected calls. Examples of such calls are `4 is 2+X` and `X>1`. The precise definition is found in section 3.1.

proof cases, but the additional complexity given by an equivalence relation (and state invariant; below) may in some cases require an unfolding into an infinite number of subcases, each requiring a differently shaped proof diagram.

The notion of observable confluence [15] for CHR considers only states that satisfy a given invariant. We include such invariants, as we consider them to be central in CHR programming practice: a program is typically developed with a particular class of queries in mind, often strongly biased, so only queries in this class lead to meaningful computations.

**Example 2.** (*Example 1, continued*) The one-line program above reflects a tacitly assumed state invariant: only one `set` constraint is allowed. If we open up for a query such as

```
?- item(a), item(b), set([]), set([c]).
```

we obtain a collection of different answers, representing different ways of splitting  $\{a, b, c\}$  into two disjoint subsets. However, this may not be intended, and the program is not confluent modulo the indicated equivalence relation unless the invariant is taken into account. The relevant invariant may specify that all constraints must be ground, and that a state must include exactly one `set/2` constraints whose argument is a list.

The earlier approach [15] for showing observable confluence (for logical built-ins only) sticks to the above mentioned logical subsumption principle. As shown by [15] and explained below, this leads to infinitely many proof cases for even simple invariants such as groundedness; our meta-language approach handles such examples in a more satisfactory way.

Confluence modulo equivalence was mentioned in relation to CHR in a previous conference paper [12] that also gave a first version of the operational semantics. The present paper provides theoretical foundations for studying confluence modulo equivalence for CHR, and introduces a formal meta-language that supports systematic proofs. This may also point forward towards (partly) mechanized proof systems for confluence modulo equivalence.

The results in the present paper may carry over in a useful way to other systems with nondeterminism in which confluence has to be studied. This may be active rules in databases [6], concurrent constraint programming [17] and theoretical models of concurrency such as  $\pi$ - and  $\rho$ -calculi [31, 32].

Section 2 reviews previous work on confluence in term rewriting and general rewriting systems, including fundamental results concerning confluence modulo equivalence, that has not been utilized for CHR before, and we give an overview of the state of the art for CHR. Section 3 gives our operational semantics for CHR, first introduced in [12], intended for reasoning about confluence for programs with non-logical built-in predicates, and various properties related to confluence are introduced; we also make a comparison with operational semantics used in earlier work on confluence for CHR. In Section 4 we generalize earlier results on critical pairs for CHR, now including the larger set of built-in predicates, and taking invariant and equivalence into account; we can also show that such pairs – or corners as we call them (since we include the common ancestor state) – are not suited for proofs of confluence in our more general case due to this subsumption principle; we also add some more detailed comments on previous work on confluence for CHR.

Our main results are presented in Section 5. The meta-language METACHR is introduced in which proofs of joinability are reified as abstract diagrams. A proof of confluence modulo equivalence can be split into a finite set of proof cases, each given by an abstract corner. As opposed to the results of [1, 3] it is not necessary for confluence (modulo equivalence) that each such abstract corner is joinable. A property called split-joinable is introduced, occasionally leading to infinite sets of corners to be checked for joinability. We show that when the abstract corners are either joinable or split-joinable, local confluence is guaranteed and confluence is guaranteed for terminating programs.

In Section 6, we demonstrate the applicability of the suggested approach, by giving proofs of confluence modulo equivalence for selected programs: the program of Example 1 that demonstrates an equivalence indicating a redundant data representation, a version of the Viterbi algorithm in CHR that exemplifies dynamic programming algorithms with pruning, and finally an example with a splitting into infinitely many cases. Section 7 provides for a summary, and a discussion of possible directions for future work.

## 2. Background and Related work

Confluence modulo trivial identity is well-studied in Rewriting Systems, see, e.g., [8] for an overview. Since the 1990es, the proof methods have been adapted to the more complex system of Constraint Handling Rules [21, 22], most notably [1, 3, 15]. Confluence modulo equivalence has been studied in general rewriting systems [26] and was only recently introduced to CHR [12].

### 2.1. Confluence for General Rewriting Systems and Term Rewriting Systems

A binary *relation*  $\rightarrow$  on a set  $A$  is a subset of  $A \times A$ , where  $x \rightarrow y$  denotes membership of  $\rightarrow$ . A *rewriting system* is a pair  $\langle A, \rightarrow \rangle$ ; it is *terminating* if there is no infinite chain  $a_0 \rightarrow a_1 \rightarrow \dots$ . The *reflexive transitive closure* of  $\rightarrow$  is denoted  $\rightarrow^*$ . The *inverse relation*  $\leftarrow$  is defined by  $\{(y, x) \mid x \rightarrow y\}$ . An *equivalence (relation)*  $\approx$  is a binary relation on  $A$  that is reflexive, transitive and symmetric. We say that  $x$  and  $y$  are *joinable* if there exists a  $z$  such that  $x \rightarrow^* z$  and a  $z \leftarrow^* y$ .

A rewriting system  $\langle A, \rightarrow \rangle$  is *confluent* if and only if  $y' \leftarrow^* x \rightarrow^* y \Rightarrow \exists z. y' \rightarrow^* z \leftarrow^* y$ , and is *locally confluent* if and only if  $y' \leftarrow x \rightarrow y \Rightarrow \exists z. y' \rightarrow^* z \leftarrow^* y$ . In 1942, Newman showed his fundamental Lemma [30]: *A terminating rewriting system is confluent if and only if it is locally confluent.* An elegant proof of Newman's lemma was provided by Huet [26] in 1980.

The more general notion of *confluence modulo equivalence* was introduced in 1972 by Aho et al [5] in the context of the Church-Rosser property.

**Definition 1 (Confluence modulo equivalence).** A relation  $\rightarrow$  is confluent modulo an equivalence  $\approx$  if and only if

$$\forall x, y, x', y'. \quad y' \leftarrow^* x' \approx x \rightarrow^* y \quad \Rightarrow \quad \exists z, z'. \quad y' \rightarrow^* z' \approx z \leftarrow^* y.$$

Given an equivalence relation  $\approx$ , we say that  $x$  and  $y$  are *joinable modulo equivalence* if there exists  $z, z'$  such that  $x \rightarrow^* z, z' \leftarrow^* y$  and  $z \approx z'$ . This is shown as a diagram in Fig. 1a. In 1974, Sethi [38] studied confluence modulo equivalence for bounded rewriting systems, that are systems for which there exists an upper bound for the number of possible rewrite steps for all terms. He showed that confluence modulo equivalence for bounded systems is equivalent to the following properties,  $\alpha$  and  $\beta$ , also shown in Fig. 1b.

**Definition 2 ( $\alpha$  &  $\beta$ ).** A relation  $\rightarrow$  has the  $\alpha$  property and the  $\beta$  property with respect to an equivalence  $\approx$  if and only if it satisfies the  $\alpha$  and  $\beta$  conditions, respectively:

$$\begin{aligned} \alpha : \quad & \forall x, y, y'. \quad y' \leftarrow x \rightarrow y \quad \Rightarrow \quad \exists z, z'. \quad y' \rightarrow^* z' \approx z \leftarrow^* y \\ \beta : \quad & \forall x, y', y. \quad y' \approx x \rightarrow y \quad \Rightarrow \quad \exists z, z'. \quad y' \rightarrow^* z' \approx z \leftarrow^* y \end{aligned}$$

In 1980, Huet [26] generalized this result to any terminating system.

**Definition 3 (Local confl. mod. equivalence).** A rewriting system is *locally confluent modulo an equivalence*  $\approx$  if and only if it has the  $\alpha$  and  $\beta$  properties.

**Theorem 1. (Huet, [26])** Let  $\rightarrow$  be a terminating rewriting system. For any equivalence  $\approx$ ,  $\rightarrow$  is confluent modulo  $\approx$  if and only if  $\rightarrow$  is locally confluent modulo  $\approx$ .

Term rewriting systems have been studied extensively, and terminology and several important results carry over to CHR, as we will see below. In the following, we assume the reader familiar with the notions of terms over some signature and variables, substitutions and most general unifiers.

**Definition 4 (Term Rewriting System; semi-formal version adapted from [8]).** A *term rewriting system (TRS)* consists of a finite set of rules of the form  $(l, r)$  in which any variable in  $r$  also appears in  $l$ . The application of such a rule to a term  $s$  to obtain another term  $t$ , written  $s \rightarrow t$  is obtained by 1) find a substitution  $\theta$ , such that  $l\theta$  is a subterm of  $s$ , and 2)  $t$  is given by replacing that subterm in  $s$  by  $r\theta$ .

The following notion of critical pairs represents cases in which two rules both can apply in the same subterm, but if one is applied, the second one cannot be applied successively.

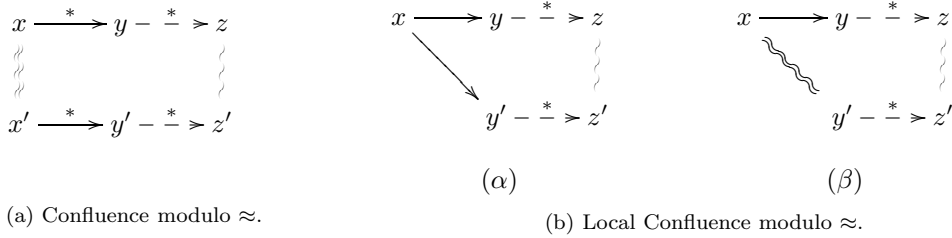


Fig. 1. Diagrams for the fundamental notions. A dotted arrow (single wave line) indicates an inferred step (inferred equivalence).

**Definition 5 (TRS Critical Pair; adapted from [8]).** Consider  $R_k = (l_k, r_k), k = 1, 2$  (assumed re-named apart so they have no variable in common) for which there is a most general unifier  $\sigma$  of  $l_2$  and a non-variable subterm of  $l_1$ . Then  $\langle t_1, t_2 \rangle$  is a *critical pair*, whenever  $l_1\sigma \rightarrow t_k$  using  $R_k, k = 1, 2$ .

For example, the two rules  $(f(a), b)$  and  $(a, c)$  give rise to the critical pair  $\langle b, f(c) \rangle$ ; both are derived from the common ancestor term  $f(a)$ , i.e.,  $b \leftarrow f(a) \rightarrow f(c)$ .

In 1970, Knuth and Bendix [27] developed the following, fundamental properties, later elaborated by Huet [26]. We bring them in detail as very similar properties holds for CHR.

**Lemma 1 (Critical Pair Lemma for TRS [26, 27]).** Let a TRS be given and assume terms  $s, t_1, t_2$  such that  $t_1 \leftarrow s \rightarrow t_2$ . Then either

- $t_1$  and  $t_2$  are joinable, or
- there exists an instance  $\langle u_1, u_2 \rangle$  or  $\langle u_2, u_1 \rangle$  of a critical pair and a specific subterm  $s'$  of  $s$  such that  $t_k$  is a copy of  $s$  in which  $s'$  is replaced by  $u_k, k = 1, 2$ .

**Theorem 2 (Critical Pair Theorem for TRS [26, 27]).** A TRS is locally confluent if and only if all its critical pairs are joinable.

This theorem in combination with Newman's lemma leads to a desired result: *A terminating TRS is confluent if and only if all its critical pairs are joinable*. Furthermore, confluence of a finite terminating TRS is decidable (as there is only a finite number of critical pairs and finitely many finite derivations to test out from their states).

Mayr and Nipkow [29] studied confluence modulo equivalence for a subset of higher-order rewriting systems (that extend term rewriting to  $\lambda$ -terms). They used an alternative version of Theorem 1 in which the  $\beta$  property is replaced by a  $\gamma$  property, as shown below. It applies when the equivalence  $\approx$  is specified as the transitive closure of a symmetric relation  $\vdash$ ; such a relation may, e.g., be generated by a set of equations.

**Lemma 2 ( $\alpha$  &  $\gamma$  Confluence [26]).** Let  $\vdash$  be a symmetric relation and  $\approx = (\vdash)^*$ . Let  $\rightarrow$  be any relation such that the composition  $\rightarrow \cdot \approx$  is terminating. Then  $\rightarrow$  is confluent modulo  $\approx$  if and only if the conditions  $\alpha$  and  $\gamma$  are satisfied:

$$\begin{aligned} \alpha : \quad & \forall x, y, y'. \quad y' \leftarrow x \rightarrow y \quad \implies \quad \exists z, z'. \quad y' \xrightarrow{*} z' \approx z \xleftarrow{*} y \\ \gamma : \quad & \forall x, y', y. \quad y' \vdash x \rightarrow y \quad \implies \quad \exists z, z'. \quad y' \xrightarrow{*} z' \approx z \xleftarrow{*} y \end{aligned}$$

We do not use this lemma in the present paper, but possible applications are discussed in the concluding section.

## 2.2. Confluence for Constraint Handling Rules

Constraint Handling Rules, CHR, can be understood as a rewrite system over states that are multisets of constraints as shown in Example 1 above, p. 2.<sup>4</sup>

The known results on confluence for CHR are very similar to those on term rewriting systems shown above. Similar critical pairs of states may appear when two instances of rules can apply to overlapping constraints; the precise definition is given in Section 3 below. The following shows the construction of such a critical pair for an overlap of two different instances of the only rule in the program of Example 1, p. 2, above.

$$\{\text{item}(Y), \text{set}([X|L])\} \leftarrow \{\text{item}(X), \text{item}(Y), \text{set}(L)\} \rightarrow \{\text{item}(X), \text{set}([Y|L])\}$$

The first publications by Frühwirth on CHR appeared in 1993–4 [19, 20]. Soon after, around 1996, the central results on confluence for CHR were developed by Abdennadher and others [1, 3], however, only for the subset of CHR with logical built-ins and neither invariant nor equivalence. The concepts and results from the area of term rewriting can be transferred to CHR so that the following results hold;  $\text{CHR}^0$  refers to the indicated subset of CHR.

- A  $\text{CHR}^0$  program is locally confluent if and only if all its critical pairs are joinable.
- The set of critical pairs is finite and local confluence is decidable; automatic checkers of this property has been developed for  $\text{CHR}^0$ , e.g., [28]
- A terminating  $\text{CHR}^0$  program is confluent if and only if all its critical pairs are joinable.

These results are based on the previously mentioned subsumption principle which essentially boils down to the following.

- (\*) whenever a (e.g., critical) pair of  $\text{CHR}^0$  states  $x, y$  are joinable, it holds for any substitution  $\theta$  and constraint set  $s$  that  $x\theta \cup s$  and  $y\theta \cup s$  are joinable.

Section 4.2, p. 15, gives a precise analysis and also shows that these results do not generalize directly to the larger subset of CHR considered in the present paper.

In 2007, Duck et al [15] argued for the introduction of state invariants; a state invariant  $I(\cdot)$  is a property that is preserved by the derivations of the current program, and it may, e.g., be defined by reachability from a set of intended queries. We define an *I-state*  $x$  as a state for which  $I(x)$  holds. The precise definitions and arguments are given in Section 3.2, respectively 4.2. They define (*local*) *observable confluence* for  $\text{CHR}^0$  as above, considering only derivations between *I-states*.

While this generalization of confluence is highly relevant from a practical point of view, it is inherently more difficult, as the property (\*) above does not generalize. For this discussion, we refer to a state  $x\theta \cup s$  (pair  $\langle x\theta \cup s, y\theta \cup s \rangle$ ) as an *extension* of state  $x$  (pair  $\langle x, y \rangle$ ). A state  $x$  (e.g., in a critical pair) may not be an *I-state* in itself, but some of its extended states may be *I-states*; the other way round, some extensions of an *I-state* may not be *I-states*. Duck et al [15] considered cases where, for each critical pair  $\langle x, y \rangle$ , a collection of most general extensions  $\{\langle x_i, y_i \rangle\}_{i \in I_{nx}}$  exists, such that any such  $\langle x_i, y_i \rangle$  and any extension of it consists of *I-states*.

For a given program  $\Pi$  and invariant  $I$ , let  $\mathcal{M}^{I, \Pi}$  be the set of all such most general extensions for all critical pairs. Then the following holds.

- A  $\text{CHR}^0$  program is locally observably confluent w.r.t.  $I$  if and only if all pairs in  $\mathcal{M}^{I, \Pi}$  are joinable.
- A terminating  $\text{CHR}^0$  program is observably confluent w.r.t.  $I$  if and only if all pairs in  $\mathcal{M}^{I, \Pi}$  are joinable.

Decidability is lost, and [15] shows that even a standard invariant such as groundedness leads to infinite  $\mathcal{M}^{I, \Pi}$  sets. The characterization of  $\mathcal{M}^{I, \Pi}$  is complicated, and no practically relevant methods have been proposed. In the present paper, we cope with these problems by introducing a meta-language in which we can reason about abstract versions of critical pairs and their joinability, and in which the invariant is treated as a meta-level constraint.

We are not aware of other work than our own on confluence for CHR that includes non-logical predicates or takes an equivalence relation into account. Confluence for nonterminating CHR programs has been studied

---

<sup>4</sup> The rule in the example program is a so-called simplification rule. CHR also includes other types of rules, that do not introduce additional conceptual difficulties in relation to confluence, although they imply an extra notational overhead.

by [23, 34], and [2] has considered how the integration of two programs known to be confluent can be made confluent by adding new rules.

The choice of an operational semantics for CHR, i.e., a definition of the derivation relation for CHR, influences the set of programs recognized as confluent and the amount of notational overhead needed for the proofs. We postpone a comparison with selected other operational semantics until Section 3.3, following the introduction of the necessary technical apparatus.

### 3. Constraint Handling Rules

In the following, we introduce CHR and our new operational semantics as a rewriting system. We highlight the differences in comparison with previous semantics used for the study of confluence for CHR. Ours differs most essentially in that it can describe non-logical and incomplete built-ins, and we have also succeeded in introducing several simplifications without loss of generality (apart from a subtle mathematical consequence implied by some earlier semantics exposed in Example 12, p. 18).

#### 3.1. Preliminaries

We extend the basic concepts and notation introduced in Section 2.1. Derivation steps are labelled so we can distinguish how they are produced with reference to the CHR program in question (letters  $D$  and  $d$  are typically used for such labels, indicating a *description* of the step). We also introduce the notions  $\alpha$ - and  $\beta$ -corners to give a representation of cases where the  $\alpha$ - and  $\beta$  conditions may (or may not) hold.

**Definition 6.** A *derivation system*  $\langle S, D, \mapsto, I, \approx \rangle$  consists of a set  $S$ , called *states*, a set of *labels*  $D$ , a ternary *derivation relation*  $\mapsto \subseteq S \times D \times S$ , an *invariant*  $I \subset S$ , and an *equivalence*  $\approx \subseteq S \times S$ .

A fact  $\langle x, d, y \rangle \in \mapsto$  is written  $x \xrightarrow{d} y$ , in which case we also write  $x \mapsto y$ , thus projecting it to a binary relation; as usual  $\mapsto^*$  denotes the reflexive, transitive closure of  $\mapsto$ , and *derivation* is a successive sequence of zero or more, perhaps infinitely many, derivation steps. For brevity, we may use  $x \xrightarrow{*} y$  to indicate a derivation from  $x$  to  $y$ , with labels understood. The invariant property of  $I$  means that  $I(x) \wedge (x \xrightarrow{d} y)$  implies  $I(y)$ ; a state  $x$  with  $I(x)$  is an *I-state* and *I-derivation* (step)s are those that involve only *I-states*.

An  $\alpha$ -*corner* is a structure of the form  $y' \leftarrow x \mapsto y$  where  $x, y, y'$  are states and  $y' \leftarrow x, x \mapsto y$  are derivation steps; a  $\beta$ -*corner* is of the form  $y' \approx x \mapsto y$  where  $x, y, y'$  are states,  $y' \approx x$  holds and  $x \mapsto y$  is a derivation step. We may use the symbol  $\Delta$  to denote a corner. In both cases, the state  $x$  is referred to as the *common ancestor state* for the *wing states*  $y'$  and  $y$ . Two  $\alpha$ -corners  $y' \leftarrow x \mapsto y$  and  $y \leftarrow x \mapsto y'$  are considered identical. An  $\alpha$ - ( $\beta$ -) corner is called an  $\alpha$ - ( $\beta$ -) *I-corner* when its states are *I-states*.

A *joinability diagram* (modulo  $\approx$ ) for an  $\alpha$ - or  $\beta$ -corner

$$y' \text{ Rel } x \xrightarrow{d_2} y$$

(thus *Rel* is one of  $\leftarrow^*$  or  $\approx$ ) is a structure of the form

$$z' \leftarrow^* y' \text{ Rel } x \xrightarrow{d_3} y \xrightarrow{*} z$$

where  $z' \leftarrow^* y'$  and  $y \xrightarrow{*} z$  are derivations such that the equivalence  $z' \approx z$  holds. A diagram is sometimes denoted by the symbol  $\Delta$ . A given corner is *joinable* modulo  $\approx$  whenever there exists a joinability diagram for it. An  $\alpha$ -corner of the form  $y \leftarrow x \mapsto y$  is called *trivially joinable* (modulo  $\approx$ ).

A derivation system  $\langle S, D, \mapsto, I, \approx \rangle$  is *confluent modulo  $\approx$*  (with respect to  $I$ ) if and only if, for all *I-states*  $y', x, y$ :  $y' \leftarrow^* x \xrightarrow{*} y \Rightarrow \exists z, z'. y' \xrightarrow{*} z' \approx z \leftarrow^* y$ , and is *locally confluent modulo  $\approx$*  (with respect to  $I$ ) if and only if all its *I-corners* are joinable modulo  $\approx$ .

Joinability diagrams may be shown as in Figure 1b, and notions of (local) (*I*-) confluence (modulo  $\approx$ ) *I*-termination apply as already introduced. We can reformulate Theorem 1 as follows.

**Theorem 3.** An *I*-terminating derivation system is *I*-confluent modulo  $\approx$  if and only if all its *I-corners* (of type  $\alpha$  as well as  $\beta$ ) are joinable modulo  $\approx$ .

We assume standard notions of first-order logic such as predicates, atoms and terms. For any expression  $E$ ,  $\text{vars}(E)$  refers to the set of variables occurring in  $E$ . A *substitution* is a mapping from a finite set of variables to terms, e.g., the substitution  $[x/t]$  replaces variable  $x$  by term  $t$ . For substitution  $\sigma$  and expression  $E$ ,  $E\sigma$  (or  $E \cdot \sigma$ ) denotes the expression that arises when  $\sigma$  is applied to  $E$ ; composition of two substitutions  $\sigma, \tau$  is denoted  $\sigma \circ \tau$ . Special substitutions *failure* and *error* are assumed, the first one representing falsity and the second one runtime errors; a substitution different from these two is called a *proper substitution*.

Two disjoint sets of (*user*) *constraints* and *built-in* predicates are assumed. Our semantics for built-ins differs from previous approaches by mapping them immediately to a unique substitution. This makes it possible to handle non-logical devices such as Prolog's `var/1` and run-time errors as they may arise from incomplete built-ins such as `is/2`.

An evaluation procedure  $\text{Exe}$  for built-in atoms  $b$  is assumed, such that  $\text{Exe}(b)$  is either a (possibly identity) substitution to a subset of  $\text{vars}(b)$  or one of *failure* and *error*. It extends to sequences of built-ins as follows.

$$\text{Exe}((b_1, b_2)) = \begin{cases} \text{Exe}(b_1) & \text{when } \text{Exe}(b_1) \in \{\text{failure}, \text{error}\}, \\ \text{Exe}(b_2 \cdot \text{Exe}(b_1)) & \text{when otherwise } \text{Exe}(b_2 \cdot \text{Exe}(b_1)) \in \{\text{failure}, \text{error}\}, \\ \text{Exe}(b_1) \circ \text{Exe}(b_2 \cdot \text{Exe}(b_1)) & \text{otherwise} \end{cases}$$

A built-in  $b$  or sequence of such is *satisfiable* whenever there exists a substitution  $\theta$  such that  $\text{Exe}(b\theta)$  is a proper substitution. A subset of built-in predicates are the *logical* ones, whose meaning is given by a first-order theory  $\mathcal{B}$ . For a logical atom  $b$  with  $\text{Exe}(b) \neq \text{error}$ , the following conditions must hold.

- Partial correctness:  $\mathcal{B} \models \forall_{\text{vars}(b)} (b \leftrightarrow \exists_{\text{vars}(\text{Exe}(b)) \setminus \text{vars}(b)} \text{Exe}(b))$ .
- Instantiation monotonicity:  $\text{Exe}(b \cdot \sigma) \neq \text{error}$  for all substitutions  $\sigma$ .

A built-in predicate  $p$  is *incomplete* if there exists an atom  $b$  with predicate  $p$  for which  $\text{Exe}(b) = \text{error}$ ; any other built-in predicate is *complete*. Any built-in predicate which is not logical is called *non-logical*. A *most general instance* of a built-in predicate  $p/n$  is an atom  $p(v_1, \dots, v_n)$  where  $v_1, \dots, v_n$  are new and unused variables. The following predicates are examples of built-ins, and the list can be extended if needed.

**Definition 7.** The following list of built-in predicates are assumed with their meaning as indicated;  $\epsilon$  is the identity substitution.

1.  $\text{Exe}(t = t') = \sigma$  where  $\sigma$  is a most general unifier of  $t$  and  $t'$ ; if no such unifier exists, the result is *failure*.
2.  $\text{Exe}(\text{true})$  is  $\epsilon$ .
3.  $\text{Exe}(\text{fail})$  is *failure*.
4.  $\text{Exe}(t \text{ is } t') = \text{Exe}(t = v)$  whenever  $t'$  is a ground term that can be interpreted as an arithmetic expression with value  $v$ ; if no such  $v$  exists, the result is *error*.
5.  $\text{Exe}(t \geq t')$  is  $\epsilon$  whenever  $t, t'$  are ground terms that can be interpreted as arithmetic expressions with values  $v, v'$  where  $v \geq v'$ ; if such values exist but  $v < v'$ , the result is *failure*; otherwise, the result is *error*.
6.  $\text{Exe}(\text{var}(t))$  is  $\epsilon$  if  $t$  is a variable and *failure* otherwise.
7.  $\text{Exe}(\text{nonvar}(t))$  is  $\epsilon$  when  $t$  is not a variable and *failure* otherwise.
8.  $\text{Exe}(\text{ground}(t))$  is  $\epsilon$  when  $t$  is ground and *failure* otherwise.
9.  $\text{Exe}(\text{constant}(t))$  is  $\epsilon$  when  $t$  is a constant and *failure* otherwise.
10.  $\text{Exe}(t == t')$  is  $\epsilon$  when  $t$  and  $t'$  are identical and *failure* otherwise.
11.  $\text{Exe}(t \setminus = t')$  is  $\epsilon$  when  $t$  and  $t'$  are non-unifiable and *failure* otherwise.

The first three predicates in Definition 7 above are logical and complete; “`is`” and “`>=`” are logical but not complete. The remaining ones are non-logical.

For the representation of CHR execution states, we introduce *indices*: an *indexed set*  $S$  is a set of items of the form  $i:x$  where  $i$  belongs to some index set and each such  $i$  is unique in  $S$ . When clear from context, we may identify an indexed set  $S$  with its cleaned version  $\{x \mid i:x \in S\}$ . Similarly, the item  $x$  may identify the indexed version  $i:x$ . We extract the indices by  $\text{id}(i:x) = i$ .



### 3.2. Operational Semantics

The following operational semantics is based on principles introduced in [12]; it differs from those used in previous work in several ways that we discuss in Section 3.3 below.

As custom in recent theoretical work on CHR, we use the *generalized simpagation* form [22] as a common representation for the rules of CHR. The guards can modify variables that also occur in rule bodies, but not variables that occur in the constraints matched by the head rules.

**Definition 8.** A rule  $R$  is of the form

$$r: H_1 \setminus H_2 \leq=> g \mid C,$$

where  $r$  is a unique identifier for the rule,  $H_1$  and  $H_2$  are sequences of constraints, forming the *head* of the rule,  $g$  is a *guard* being a sequence of built-ins, and  $C$  is a sequence of constraints and built-ins called the *body* of  $R$ . Any of  $H_1$  and  $H_2$ , but not both, may be empty. A *program* is a finite set of rules.

A *most general pre-application instance* of rule  $R$  is an indexed variant  $R'$  of  $R$  containing new and fresh variables.

An *application instance* of rule  $R$  is a structure of the form

$$R'' = R'\sigma = (r: H'_1\sigma \setminus H'_2\sigma \leq=> g'\sigma \mid C'\sigma)$$

where  $R'$  is a most general pre-application instance,  $\sigma$  is a substitution for the variables of  $H'_1, H'_2$  and  $Exe(g'\sigma)$  is a proper substitution such that<sup>5</sup>

$$(H'_1 \uplus H'_2)\sigma = (H'_1 \uplus H'_2)\sigma Exe(g'\sigma).$$

The part  $g' (g'\sigma)$  is referred to as the *guard of  $R'$*  ( $R''$ ). The *application record* for  $R'$  ( $R''$ ), denoted  $\text{applied}(R')$  ( $\text{applied}(R'')$ ) is the structure

$$r @ i_1 \dots i_n$$

where  $i_1 \dots i_n$  is the sequence of indices of  $H_1, H_2$  in the order they occur.

A rule is a *simplification* when  $H_1$  is empty, a *propagation* when  $H_2$  is empty; in both cases, the backslash is left out, and for a propagation, the arrow symbol is written  $==>$  instead. Any other rule is a *simpagation*.

Following [33], an execution state is defined in terms of a suitable equivalence class that abstracts away irrelevant details concerning which actual variables and indices are used.

**Definition 9.** A *(CHR) state representation* is a pair  $\langle S, T \rangle$ , where

- $S$  is a finite, indexed set of atoms called the *constraint store*,
- $T$  is a set of relevant application records called the *propagation history*,

where a *relevant* application record is one in which each index refers to an index in  $S$ . Two state representations  $S_1$  and  $S_2$  are *variants*, denoted  $S_1 \equiv S_2$ , whenever one can be obtained from the other by a renaming of variables and a consistent replacement of indices (i.e., by a 1-1 mapping). When  $\Sigma$  is the set of all state representations, a *(CHR) state* is an element of  $\Sigma / \equiv \cup \{\text{failure}, \text{error}\}$ , i.e., an equivalence class in  $\Sigma$  induced by  $\equiv$  or one of two special states; applying the *failure* (*error*) substitution to a state yields the *failure* (*error*) state. To indicate a given state, we may for simplicity mention one of its representations. A state different from *failure* and *error* is called a *proper state*. A *query*  $q$  is a conjunction of constraints, which is also identified with an initial state  $\langle q', \emptyset \rangle$  where  $q'$  is an indexed version of  $q$ .

Assuming a fixed program, the function *all-relevant-app-recs* from constraint stores to the powerset of application records is defined as

$$\begin{aligned} \text{all-relevant-app-recs}(S) = \{ & r @ i_1 \dots i_n \mid r \text{ identifies a propagation rule and} \\ & i_1 \dots i_n \text{ are indices of constraints to which the rule can apply} \} \end{aligned}$$

To simplify notation when we make statements involving several states or other entities involving components of states, we may do so referring to selected state representations, considering recurrence of indices and

<sup>5</sup> The condition indicates that the guard's substitution is not allowed to instantiate the variables in the head part.

variables significant. For example, in the context of a program that includes the rule  $r: p \Rightarrow q$ , we consider the following as a true statement.

$$ST = \langle \{1:p, 2:q\}, \emptyset \rangle \wedge ST = \langle S, \emptyset \rangle \wedge \text{all-relevant-app-recs}(S) = \{r@1\}$$

**Definition 10.** A *derivation step*  $\mapsto$  from one  $I$ -state to another can be of two types: by rule application instance  $\xrightarrow{R}$  or by built-in  $\xrightarrow{b}$ , defined as follows.

**Apply:**  $\langle S \uplus H_1 \uplus H_2, T \rangle \xrightarrow{R} \langle S \uplus H_1 \uplus (C \cdot \text{Exe}(g)), T' \rangle$   
 whenever there is an application instance  $R$  of the form  $r: H_1 \setminus H_2 \Leftarrow g \mid C$  with  $\text{applied}(R) \notin T$ , and  $T'$  is derived from  $T$  by 1) removing any application record having an index in  $H_2$  and 2) adding  $\text{applied}(R)$  in case  $R$  is a propagation.

**Built-in:**  $\langle \{b\} \uplus S, T \rangle \xrightarrow{b} \langle S, T \rangle \cdot \text{Exe}(b)$ .

Notice that the removal of application records in **Apply** steps ensures that no non-relevant propagation record remains in the new state (i.e., the result *is* a state).

**Example 3.** Consider a program consisting of the following two rules.

$r_1: p(X) \setminus q(Y) \Leftarrow X=Y \mid r(X)$ .  
 $r_2: r(X) \Rightarrow s(X)$ .

The following is an application instance of  $r_1$ .

$$R_1^{a,a} = (r_1: 1:p(a) \setminus 2:q(a) \Leftarrow a=a \mid 3:r(a))$$

It can be used in an **Apply** derivation step as follows.

$$\langle \{1:p(a), 2:q(a)\}, \emptyset \rangle \xrightarrow{R_1^{a,a}} \langle \{1:p(a), 3:r(a)\}, \emptyset \rangle$$

However, the indexed instance of  $r_1$ ,  $(r_1: 1:p(Z) \setminus 2:q(a) \Leftarrow Z=a \mid 3:r(Z))$  is not an application instance as the guard, when executed, will bind the head variable  $Z$ .

The rule  $r_2$  is a propagation rule, and we show an application instance for it and an **Apply** derivation step; here the propagation history is checked before the step and modified by the step.

$$R_2^a = (r_2: 1:r(a) \Rightarrow 4:s(a))$$

$$\langle \{1:r(a), 2:r(b), 3:s(b)\}, \{r_2@2\} \rangle \xrightarrow{R_2^a} \langle \{1:r(a), 2:r(b), 3:s(b), 4:s(a)\}, \{r_2@2, r_2@1\} \rangle$$

The following example shows how an incomplete predicate is treated when occurring in a guard and when executed by a **Built-in** step.

**Example 4.** Consider a program that includes the following rule having the incomplete “is” predicate in its guard. Furthermore, assume that “is” can appear in **Built-in** steps, i.e., can also appear in a state.

$r_1: p(X) \Rightarrow Y \text{ is } X+2 \mid q(Y)$ .

An attempt to **Apply** it to some state by matching the head with  $1:p(2)$  may yield the application instance

$$R_2^a = (r_1: 1:p(2) \Rightarrow Z \text{ is } 2+2 \mid 7:q(Z)).$$

The guard evaluates to the substitution  $[Z/4]$  and the new state includes the instantiated body constraint  $7:q(4)$ . The rule cannot apply by matching the head with  $2:p(Z)$  as the guard evaluates to the *error* substitution – but no *error* state is produced. A **Built-in** step, on the other hand, for  $Z \text{ is } 2+A$  leads to the *error* substitution (by Definition 7) and in turn to the *error* state.

We observe the following immediate consequence of the definition, namely a functional dependency from a state plus label of a possible step to the resulting state.

**Proposition 1.** For any state  $\Sigma$  and derivation step label  $d$ , there is at most one state  $\Sigma'$  such that  $\Sigma \xrightarrow{d} \Sigma'$ .

The following distinctions become useful later when we reason about derivation steps and the built-ins involved. As it appears in Definition 10 above, built-ins evaluating to *error* (representing runtime error) are treated differently in the two sorts of derivation steps: in a guard, *error* and *failure* both means that a rule cannot apply (corresponding to no runtime error reported in an implemented system); when such a built-in (coming from the query or a rule body) is applied to a state, it gives rise to a derivation step leading to the relevant of an *error* or a *failure* state.

**Definition 11.** In the context of a state invariant  $I$ , a built-in predicate is a *state built-in predicate* whenever it can appear in an  $I$ -state. A logical built-in predicate  $p$  is  *$I$ -complete* whenever  $\text{Exe}(b) \neq \text{error}$  for any atom  $b$  with predicate  $p$  that may occur in an  $I$ -state or in the guard of an application instance that can apply to an  $I$ -state.

A guard in a rule is *logical* if it contains only logical predicates; otherwise, it is *non-logical*. A logical guard is  *$I$ -complete* if it contains only  $I$ -complete predicates; otherwise, it is  *$I$ -incomplete*.

**Example 5.** The built-in “`is/2`” is logical and, while incomplete, it is  $I$ -complete with respect to an invariant that guarantees the second argument to be a ground arithmetic expression.

### 3.3. A Few Comments on Earlier Operational Semantics for CHR

Our operational semantics for CHR differs from other known and formally specified ones e.g., [1, 3, 4, 14, 15] by handling also non-logical and incomplete built-ins.

We do that by “executing” built-ins immediately in terms of substitutions applied to the state, which we claim is more compatible with practical CHR systems than earlier approaches; Apt et al’s semantics for Prolog with such predicates [7] applies the same principles (with the small difference that they do not distinguish between error and failure). The referenced approaches use instead a separate store for built-in constraints (restricted to logical ones) that have been processed, with their satisfiability determined by a magic solver that mirrors a first-order semantics; this excludes the possibility to consider runtime errors and non-logical and incomplete predicates. The following example highlights the difference.

**Example 6.** Assume a program that includes the following program rule, and assume that  $\geq$  is also a state built-in predicate.

$p(X) \Leftarrow X \geq 1 \mid r(X)$

We can point out the difference by the query  $A \geq 2, p(A)$ . Starting from an empty built-in store (**true**), the semantics of [1, 3, 4, 14, 15] may first “execute”  $A \geq 2$  by adding it to the built-in store and keeping  $p(A)$  as the remaining query. Then the program rule above can apply for  $p(A)$  – since the truth of the guard is implied by the built-in store, thus leaving a final constraint store  $\{r(A)\}$  constrained by the built-in store ( $A \geq 2$ ).

With our semantics, the rule cannot apply (as the guard evaluates to *error* which is treated the same way as *failure*), and evaluating  $A \geq 2$  as part of the query results in the final state *error*.

The test in our semantics (Definition 8) that prevents a rule from being applied if it otherwise would modify variables in the constraints matched by the rule head, is implicit in [1, 3, 4, 14, 15]. Consider, for example, a rule  $p(X) \Leftarrow X = a \mid \dots$  considered for the query atom  $p(A)$ , assuming a built-in store  $B$ . Here the test in the guard would amount to the condition  $B \models \forall A. A = a$ ; this is false when, say  $B$  is empty, and holds only when  $B$  implies that  $a$  is the only possible value for  $A$ .

The interpretation of runtime error in guards as failure is described by [25] for one of the first widespread CHR compilers, released with earlier versions of SICStus Prolog. The documentation for the now dominant compiler [36] embedded in recent versions of SICStus Prolog and SWI Prolog<sup>6</sup> is not explicit about this point. A test of SWI Prolog shows that a runtime error in a guard makes the entire computation terminate with an error message. While this limits the completeness results for CHR, it has been chosen for efficiency reasons (and the fact that the guard can be reformulated to obtain error as failure if necessary) [35].

Furthermore, we disregard so-called global variables defined as those that appear in the original query. The mentioned previous approaches introduce a separate state component to memorize global variables, but

<sup>6</sup> See <http://www.swi-prolog.org>; version 7 checked February 2016.

this can be shown unnecessary. Consider a query  $q(X)$ ; we translate it into  $q(X), \text{global}('X', X)$  where  $'X'$  is a constant that serves as the name of variable global variable  $X$ . When a derivation terminates in a proper state, it includes the constraint  $\text{global}('X', val)$  where  $val$  is the value computed for variable  $X$ .

The mentioned semantics uses a separate state component, that we will call the *queue*, to hold constraints that have not yet been entered into the “active” constraint store. Constraints appearing in the body of a rule being applied are first entered into the queue, and then from time to time moved into the active store by a separate sort of derivation step. Rules are applied by matching constraints within the active store. This separation may be relevant as a starting point for imposing strategies for ordering application of rules and search for constraints to be processed (which is one of the goals of [14]), but for studies of confluence it is irrelevant as the set of derivations with or without this additional mechanics is essentially the same.

Our semantics has only two state components, in comparison to, e.g., [1, 3, 15] that need five state components when considering more restricted confluence problems.

There are also differences in how to avoid the potential looping by propagation rules applying to the same constraints over and over again. Our semantics (and some others not referenced) hold a set of records telling which propagations must not apply (because they have been applied already), while [1, 3, 15] maintain a set of permissions for those propagations that may apply. There is essentially only a notational difference between the two, and the choice is a matter of taste. An alternative approach is taken by [9], mixing a set-based and a multiset-based approach: new constraints produced from the body of a propagation is treated set-wise, and a propagation is only allowed if it results in adding new constraints.

In earlier work, such as [1, 3, 15] already discussed, the states include specific indices and specific variables. Thus any reasonable definition of joinability and confluence needed to mention an equivalence relation telling two states equivalent if they differ only in systematic replacement of variables and indices (quite similar to our  $\equiv$  in Definition 9, p. 9). So in some sense, these approaches concern confluence modulo equivalence problems, but for a very specific equivalence hardcoded into the proofs of general properties.<sup>7</sup> In 2009, the paper [33] gave a satisfactory solution to this problem, abstracting away concrete indices and variables defining a state as an equivalence class modulo such a relation  $\equiv$ , exactly as we have shown in our Definition 9 above.

## 4. Confluence Modulo Equivalence for CHR

Here we adapt classical definitions of critical pairs and associated properties for CHR to include non-logical and incomplete built-ins, as well as an invariant and an equivalence relation.

For the strictly logical case with no invariant, [1] defines critical pairs consisting of CHR states that may be shown joinable by ordinary CHR derivations. This is not viable in our more general case as our analogous construction may lead to pairs that do not satisfy the invariant and from which no derivations are possible (although the set of all relevant instances thereof may be joinable at the level of CHR). As a first step towards our meta-level counterpart of critical pairs, we introduce what we call most general critical pre-corners having a CHR state serving as a common ancestor.

We use the following subcategorization, introduced by [12], of  $\alpha$ - and  $\beta$ -corners according to the sorts of derivation steps involved, as they need to be treated differently. The earlier results on confluence for CHR concern only  $\alpha_1$ -corners.

**Definition 12.** Assume a program with equivalence  $\approx$  and invariant  $I$ . Let  $\Lambda_\alpha = (y \xrightarrow{\gamma} x \xrightarrow{\delta} y')$  be an  $\alpha$ - $I$ -corner and  $\Lambda_\beta = (y \approx x \xrightarrow{\delta} y')$  a  $\beta$ - $I$ -corner.

- $\Lambda_\alpha$  is an  $\alpha_1$ - $I$ -corner whenever  $\gamma$  and  $\delta$  are rule application instances.
- $\Lambda_\alpha$  is an  $\alpha_2$ - $I$ -corner whenever  $\gamma$  is a rule application instance and  $\delta$  a built-in.
- $\Lambda_\alpha$  is an  $\alpha_3$ - $I$ -corner  $\gamma$  and  $\delta$  are built-ins.
- $\Lambda_\beta$  is a  $\beta_1$ - $I$ -corner whenever  $\delta$  is a rule application instance.
- $\Lambda_\beta$  is a  $\beta_2$ - $I$ -corner whenever  $\delta$  is a built-in.

The “- $I$ ” part of the names may be left out when clear from context.

<sup>7</sup> The same can be said about [31, 32], studying confluence in a completely different setting.

#### 4.1. Most General Critical Pre-corners

According to Proposition 1, the end state of a derivation step is functionally dependent on the initial state, and we employ this in the following definition of most general critical pre-corners in which we leave out wing states. The reason why we refer to these artefacts as *pre*-corners is that they may not be corners at all; when the wing states are attempted to be filled in, the guards or invariant may not be satisfied.

**Example 7.** Consider the following program which has non-logical guards; assume  $\approx$  being identity and  $I(\cdot) = \text{true}$ .

$r_1: p(X) \Leftarrow \text{var}(X) \mid q(X).$   
 $r_2: p(X) \Leftarrow \text{nonvar}(X) \mid r(X).$   
 $r_3: q(X) \Leftarrow r(X).$

The equality predicate  $=/2$  is here regarded as a state built-in predicate, i.e., it may appear in a query; the meaning of this and the other built-ins is given in Definition 7, p. 8. The following is an attempt to construct an  $\alpha_2$ -corner; there are no propagation rules, so we leave the empty propagation history implicit and identify states by multisets of constraints.

$$\Lambda = (\{r(Z), X=Y\} \xleftarrow{R_2^Z} \{p(Z), X=Y\} \xrightarrow{X=Y} \{p(Z)\})$$

Here  $R_2^Z$  is the rule instance  $r_2: p(Z) \Leftarrow \text{nonvar}(Z) \mid r(Z)$ ;  $R_2^Z$  is not an application instance since its guard is false, thus the hinted derivation step does not exist, and  $\Lambda$  is not a corner. However, any substitution  $\theta$  that grounds  $Z$  will lead to a corner. With  $Z\theta = a$ ,  $\Lambda\theta$  is a corner, whereas if in addition  $X\theta = b$ ,  $Y\theta = c$  we need to replace the right-most derivation step  $\dots \xrightarrow{X=Y} \{p(Z)\}$  by  $\dots \xrightarrow{b=c} \text{failure}$ .

The following definition of most general critical pre-corners is lengthy as it has one case for each sort of corners, but it is straightforward when a few elements have been explained. For  $\alpha_1$ , the common ancestor state is constructed from two rules such that the application of one prevents the subsequent application of the other. The symbol “ $\circ$ ” is an arbitrary placeholder that visually indicates the presence of some state.

Propagation histories notoriously introduce extra notation and technicalities, that are explained following the definition. We recall Definition 9, that  $\text{all-relevant-app-recs}(S)$  is the set of all application records for rules of the current program taking indices only from the constraint store  $S$ .

**Definition 13 (Most General Critical Pre-Corners).** Assume a program with equivalence  $\approx$  and invariant  $I$ .<sup>8</sup>

$\alpha_1$ : A most general critical  $\alpha_1$ -*I*-pre-corner is a structure of the form  $(\circ \xleftarrow{R_1\sigma} \langle H_1\sigma \cup H_2\sigma, T \rangle \xrightarrow{R_2\sigma} \circ)$  where

- $R_k = (r_k: A_k \setminus B_k \Leftarrow g_k \mid C_k)$ ,  $k = 1, 2$ , are two most general pre-application instances;
- let  $H_k = A_k \uplus B_k$  and assume two nonempty sets  $H'_k \subseteq H_k$  such that the set of indices used in  $H'_1$  and  $H'_2$  are identical and all other indices in  $R_1, R_2$  are unique, and let  $\sigma$  be a most general unifier of  $H'_1$  and (a permutation of)  $H'_2$ ;
- if  $r_1 = r_2$ , we must have  $A_1\sigma \neq A_2\sigma$  or  $B_1\sigma \neq B_2\sigma$ ;<sup>9</sup>
- $B_1\sigma \cap H'_2\sigma \neq \emptyset$  or  $B_2\sigma \cap H'_1\sigma \neq \emptyset$ ;
- $T = \text{all-relevant-app-recs}(H_1\sigma \cup H_2\sigma) \setminus \{r_1 @ id(A_1B_1), r_2 @ id(A_2B_2)\}$ ;
- there exists a substitution  $\theta$  such that, for  $k = 1, 2$ ,  $\text{Exe}(g_k\sigma\theta)$  is a proper substitution and  $H_k \text{Exe}(g_k\sigma\theta) = H_k$ .

$\alpha_2$ : A most general critical  $\alpha_2$ -*I*-pre-corner is a structure of the form  $(\circ \xleftarrow{R} \langle A \uplus B \uplus \{b\}, T \rangle \xrightarrow{b} \circ)$  where

- $R = (r: A \setminus B \Leftarrow g \mid C)$  is a most general pre-application instance whose guard  $g$  is non-logical or *I*-incomplete;
- there exists a substitution  $\theta$  such that  $\text{Exe}(g\theta)$  is a proper substitution,  $\text{vars}(G\theta) \cap \text{vars}(b\theta) \neq \emptyset$ , and  $H \text{Exe}(g\theta) = H$ , where  $H = A \uplus B$ ;

<sup>8</sup> Notice that only  $\alpha_2$  and  $\alpha_3$  refers to  $I$ , but we maintain the *-I*- syllable in all cases for homogeneity.

<sup>9</sup> We exclude cases with  $r_1 = r_2$  where the rule applies the same way for both derivation steps, i.e.,  $A_1\sigma = A_2\sigma$  and  $B_1\sigma = B_2\sigma$ , as the two wing states in any subsumed corner would be identical and thus trivially joinable.

- $b$  is a most general instance of a state built-in predicate (i.e., all arg's are fresh variables);
- $T = \text{all-relevant-app-recs}(A \uplus B) \setminus \{r@id(AB)\}$ .

$\alpha_3$ : A *most general critical  $\alpha_3$ -I-pre-corner* is a structure of the form  $(\circ \xleftarrow{b_1} \langle \{b_1, b_2\}, \emptyset \rangle \xrightarrow{b_2} \circ)$  where

- $b_k$ ,  $k = 1, 2$ , are indexed, most general instances of state built-in predicates,  $b_1$  being non-logical or  $I$ -incomplete.

$\beta_1$ : When  $\approx \neq$ , a *most general critical  $\beta_1$ -I-pre-corner* is a structure of the form  $(\circ \approx \langle A \uplus B, T \rangle \xrightarrow{R} \circ)$  where

- $R = (r: A \setminus B \leq g \mid C)$  is a most general pre-application instance whose guard  $g$  is satisfiable;
- $T = \text{all-relevant-app-recs}(A \uplus B) \setminus \{r@id(AB)\}$ .

$\beta_2$ : When  $\approx \neq$ , a *most general critical  $\beta_2$ -I-pre-corner* is a structure of the form  $(\circ \approx \langle \{b\}, \emptyset \rangle \xrightarrow{b} \circ)$  where

- $b$  is a most general instance of a state built-in predicate.

Any two most general critical  $I$ -pre-corners are considered the same whenever they differ only by consistent renaming of indices and variables and swapping of the left and right parts. The  $I$  part of the names may be left out when clear from context.

The propagation history constructed for  $\alpha_1$ -pre-corners is similar to that of earlier work, e.g., [1], for building critical pairs.<sup>10</sup> It tells that any other propagation rule, say *Prop*, which might accidentally be applied to constraints in the common ancestor state, is prevented from doing so. This provides the maximum level of generality of the pre-corner in the sense that it subsumes (defined below) all concrete corners in which *Prop* can apply as well as those where it cannot. The propagation histories for the other sorts of pre-corners can be explained in similar ways.

**Example 8 (continuing Example 7, p. 13).** The following is an example of a most general critical  $\alpha_2$ -pre-corner for the rule labelled  $r_1$  (whose guard contains  $\text{var}/1$ ) and built-in  $=/2$ .

$$\Lambda^{r_1,=} = (\circ \xleftarrow{R_1^Z} \langle \{p(Z), x=y\}, \emptyset \rangle \xrightarrow{x=y} \circ)$$

Here  $R_1^Z$  is the application instance  $r_1 : p(Z) \leq \text{var}(Z) \mid q(Z)$ .

As opposed to the derivation relation  $\mapsto$ , the equivalence relation is given in an atomic way, so we need to consider any possible  $\beta$ -corner as critical, i.e., its joinability is not a priori given.<sup>11</sup>

By construction, we have the following.

**Proposition 2.** For any given program with invariant  $I$  and equivalence  $\approx$ , the set of most general critical  $I$ -pre-corners is finite.

As mentioned, most general critical pre-corners are intended to provide a finite characterization of the set of actual corners that are not per se joinable. To express this, we introduce the following notion of subsumption.

**Definition 14 (Subsumption by Most General Critical Pre-Corners).** Let  $\Lambda = (\circ \text{Rel}_1 \langle S, T \rangle \text{Rel}_2 \circ)$  be a most general critical pre-corner. An  $I$ -corner  $\lambda = (\langle s_1, t_1 \rangle \text{rel}_1 \langle s, t \rangle \text{rel}_2 \langle s_2, t_2 \rangle)$  is *subsumed by*  $\Lambda$ , written  $\Lambda < \lambda$ , whenever there exists a substitution  $\theta$ , a set of indexed constraints  $s^+$  and sets of application instances  $t^+$  and  $t^\div$  such that

- $s = S\theta \uplus s^+$ ,
- $t = T\theta \uplus t^+ \setminus t^\div$ ,
- $t^+ \subseteq \text{all-relevant-app-recs}(S\theta \uplus s^+) \setminus \text{all-relevant-app-recs}(S\theta)$   
(i.e., a set of application records, each containing an index in  $s^+$ ),

<sup>10</sup> It makes only a syntactic difference that [1] maintains a set of application records for rules that may be applied, whereas we maintain a set for those that may not be applied.

<sup>11</sup> In the concluding section, we discuss an alternative approach that uses  $\gamma$ -corners, as mentioned in Section 2.1, instead of  $\beta$ -corners, which makes it possible to subcategorize and perhaps filter away some of the abstract pre-corners that concern the equivalence.

- $t^\div \subseteq \text{all-relevant-app-recs}(S\theta)$
- $\text{rel}_k = \text{Rel}_k \theta, \quad k = 1, 2.$

If, furthermore,  $\Lambda$  is an  $\alpha_2$ -pre-corner ( $\circ \xleftarrow{R} \langle \Sigma, T \rangle \xrightarrow{b} \circ$ ), where  $R$  has guard  $g$ , and  $b$  a built-in, it is required that  $\text{vars}(g\theta) \cap \text{vars}(b\theta) \neq \emptyset$ .

This definition is guilty in a slight abuse of usage due to the additional requirements for  $\alpha_2$  in that only “really critical” instances of the pre-corners are counted: if the indicated variable overlaps are not observed, the two derivation steps commute so that joinability is guaranteed.

**Example 9 (continuing Examples 7, 8).** Consider the following  $\alpha_2$ -corner for the program given in Example 7,

$$\lambda = (\langle \{q(A), A=a\} \cup S, T \rangle \xleftarrow{R_1^A} \langle \{p(A), A=a\} \cup S, T \rangle \xrightarrow{A=a} \langle \{p(a)\} \cup S[A/a], T \rangle)$$

where  $R_1^A$  is the rule instance ( $r_1 : p(A) \Leftarrow \text{var}(A) \mid q(A)$ ) and  $S$  ( $T$ ) a suitable set of indexed constraints (application records). It appears that  $\lambda$  is subsumed by the most general critical  $\alpha_2$ -pre-corner  $\Lambda^{r_1, =}$  introduced in Example 8 above. To see this, we use the substitution  $[Z/A, X/A, Y/a]$  for  $\theta$  in Definition 14 above, and check that  $\text{vars}(\text{var}(Z)\theta) = \{A\}$  and  $\text{vars}((X=Y)\theta) = \{A\}$  do overlap.

The following adapts the Critical Pair Lemma [26, 27] known from term rewriting (and implicit in previous work on confluence for CHR) to our setting.

**Lemma 3 (Critical Corner Lemma).** Assume a program with invariant  $I$  and equivalence relation  $\approx$ , and let  $\lambda$  be an  $I$ -arbitrary corner. Then it holds that either

- $\lambda$  is  $I$ -joinable modulo  $\approx$ , or
- $\lambda$  is subsumed by a most general critical pre-corner.

The proof which is straightforward but lengthy can be found in the appendix.

This leads to the following central theorem.

**Theorem 4 (Critical Corner Theorem).** Assume a program  $\Pi$  with invariant  $I$  and state equivalence relation  $\approx$ . Then  $\Pi$  is locally confluent modulo  $\approx$  if and only if all  $I$ -corners subsumed by some most general critical pre-corner for  $\Pi$  are joinable.

*Proof.* The “only if” part: Assume the opposite, that  $\Pi$  is locally confluent and that there is an  $I$ -corner  $\lambda$  subsumed by some critical pre-corner for  $\Pi$  which is not joinable modulo  $\approx$ . According to Lemma 3,  $\lambda$  must be joinable; contradiction. The “if” part follows immediately from Lemma 3: let  $\lambda$  be an  $I$ -corner; if  $\lambda$  is subsumed by some critical pre-corner for  $\Pi$  we are done by assumption; otherwise the lemma states that it is joinable.  $\square$

Combining this result with Theorem 3, p. 7, we get the following.

**Theorem 5.** Assume a terminating program  $\Pi$  with invariant  $I$  and state equivalence relation  $\approx$ . Then  $\Pi$  is confluent modulo  $\approx$  if and only if all  $I$ -corners subsumed by some critical pre-corner for  $\Pi$  are joinable.

## 4.2. Relationship with Earlier Approaches to Proving Confluence

In the following, we reformulate earlier results of Abdennadher et al [1, 3, 21] for confluence without equivalence and invariant for the purely logical subset of CHR and those of Duck et al [15], who extended with an invariant, as we have described in Section 2.2. Their critical pairs are similar to our most general critical  $\alpha_1$ -pre-corners, and the other sorts of corners become either trivially joinable or non-existing in these special cases.

In order to describe these results, we complement the notion of subsumption introduced above in Definition 14 with a subsumption ordering for  $I$ -corners.

**Definition 15 (Subsumption Ordering for  $\alpha_1$ - $I$ -corner).** Assume a program with invariant  $I$ , and let

$\lambda = (\langle s_1, t_1 \rangle \xleftarrow{r_1} \langle s_0, t_0 \rangle \xrightarrow{r_2} \langle s_2, t_2 \rangle)$  and  $\lambda' = (\langle s'_1, t'_1 \rangle \xleftarrow{r'_1} \langle s'_0, t'_0 \rangle \xrightarrow{r'_2} \langle s'_2, t'_2 \rangle)$  be  $\alpha_1$ - $I$ -corners. We say that  $\lambda$

*subsumes*  $\lambda'$  denoted  $\lambda \preceq \lambda'$  whenever there exist a substitution  $\theta_k$ , a set of indexed constraints  $s_k^+$  and sets of application instances  $t_k^+$  and  $t_k^\div$  for  $k = 0, 1, 2$  such that

- $s'_k = s_k \theta_k \uplus s_k^+$ ,
- $t'_k = t_k \uplus t_k^+ \setminus t_k^\div$ ,
- $t_k^+$  is a set of application records, each containing at least one index appearing in  $s_k^+$ ,
- $t_k^\div \subseteq \text{all-relevant-app-recs}(S_k)$
- $r'_i = r_i \theta_0$ ,  $i = 1, 2$ .

We write  $\lambda \prec \lambda'$  whenever  $\lambda \preceq \lambda'$  and  $\lambda \neq \lambda'$ .

The following property follows immediately by the direct similarity with Definition 14.

**Proposition 3.** Let  $\Lambda$  be a most general critical  $\alpha_1$ -pre-corner and  $\lambda$  an  $\alpha_1$ -corner such that  $\Lambda < \lambda$ . Whenever  $\lambda'$  is a corner with  $\lambda \preceq \lambda'$ , it holds that  $\Lambda < \lambda'$ .

**Example 10 (continuing Examples 1, p. 2, and 2, p. 3).** Consider again the single rule program that collects elements into a list, with the invariant of groundedness plus exactly one **set** constraint whose argument is a list (we ignore the state equivalence here). The following shows a most general critical pre-corner, two corners and their mutual ordering;  $R^{L,A}$  stands for an applications instance for the rule in which variables  $L, A$  are replaced by terms  $L, A$ .

$$\begin{array}{c}
 \circ \xleftarrow{R^{L,A}} \{\text{set}(L), \text{item}(A), \text{item}(B)\} \xrightarrow{R^{L,B}} \circ \\
 \wedge \\
 \{\text{set}([a|c]), \text{item}(b)\} \xleftarrow{R^{[c],a}} \{\text{set}([c]), \text{item}(a), \text{item}(b)\} \xrightarrow{R^{[c],b}} \{\text{set}([b|c]), \text{item}(a)\} \\
 \vee \\
 \{\text{set}([a|c]), \text{item}(b), \text{item}(d)\} \xleftarrow{R^{[c],a}} \{\text{set}([c]), \text{item}(a), \text{item}(b), \text{item}(d)\} \\
 \xrightarrow{R^{[c],b}} \{\text{set}([b|c]), \text{item}(a), \text{item}(d)\}
 \end{array}$$

Notice in the example above that the common ancestor in the pre-corner does not satisfy the invariant and thus there are no derivation steps possible from it. However, when this state is instantiated (and perhaps extended with more constraints) so that the invariant becomes satisfied, the derivation labels denote actual application instances and corners emerge.

We proceed now as Duck et al [15] and identify a collection of  $I$ -corners for each pre-corner which together subsumes all relevant  $I$ -corners as shown in Theorem 6 below.

**Definition 16 (Minimal and Least Critical  $I$ -corners).** Assume a program with invariant  $I$ . An  $\alpha_1$ - $I$ -corner  $\lambda$  is *minimal* (for a most general  $\alpha_1$ -pre-corner  $\Lambda$ ) whenever

- $\Lambda < \lambda$ , and
- $\nexists \lambda' > \Lambda: \lambda' \prec \lambda$ .

When, furthermore

- $\forall \lambda' > \Lambda: \lambda \preceq \lambda'$ ,

$\lambda$  is a *least  $I$ -corner* for  $\Lambda$ .

In Example 10 above, the highest placed  $I$ -corner is minimal but not least, as other similar corners exist with other choices of constants.

Theorem 6 below is similar to the central result of [15], showing that local  $I$ -confluence follows from joinability of a specific set of minimal  $I$ -corners.

**Lemma 4 (Existence of Minimal  $I$ -corners).** Assume a program with invariant  $I$ . For any  $\alpha_1$ - $I$ -corner  $\lambda'$  subsumed by a most general  $\alpha_1$ -pre-corner  $\Lambda$ , i.e.,  $\Lambda < \lambda'$ , there exists a minimal  $I$ -corner  $\lambda$  for  $\Lambda$  such that  $\Lambda < \lambda \preceq \lambda'$ .



*Proof.* First of all, we notice that by construction of subsumption, that (\*\*) there cannot exist infinite chains  $\lambda_1 \succ \lambda_2 \succ \dots \succ \Lambda$ .

Consider now  $\lambda' > \Lambda$ . If  $\lambda'$  is minimal, we are done; otherwise (by Definition 14, p. 14) there will be a  $\lambda_1$  such that  $\lambda' \succ \lambda_1 > \Lambda$ ; if  $\lambda_1$  is minimal, we are done; otherwise there will be a  $\lambda_2$  such that  $\lambda' \succ \lambda_1 \succ \lambda_2 > \Lambda$ , and we continue the same way until we reach a minimal  $\lambda_n$  with  $\lambda_1 \succ \lambda_2 \succ \dots \succ \lambda_n > \Lambda$ ; due to observation (\*\*) above, this process will terminate as indicated.  $\square$

**Lemma 5 (Minimal  $I$ -corner Lemma; logical case with invariant; trivial  $\approx$ ).**

Assume a program with logical and complete built-ins, invariant  $I$  and state equivalence  $=$ , and let  $\lambda$  be an  $\alpha_1$ - $I$ -corner. Then it holds that either

- $\lambda$  is  $I$ -joinable, or
- $\lambda$  is subsumed by some minimal  $\alpha_1$ - $I$ -corner.

*Proof.* The lemma is a direct consequence of Lemma 3 and Lemma 4.  $\square$

**Theorem 6 (Minimal  $I$ -corner Theorem; logical case with invariant; trivial  $\approx$ ).**

For a program  $\Pi$  with logical and complete built-ins, invariant  $I$  and state equivalence relation  $=$ , the following properties hold.

1.  $\Pi$  is locally confluent if and only if all its minimal  $I$ -corners are joinable.
2. When, furthermore,  $\Pi$  is terminating,  $\Pi$  is confluent if and only if all its minimal  $I$ -corners are joinable.
3. A minimal  $I$ -corner is not necessarily least, and the set of all minimal  $I$ -corners is not necessarily finite.

*Proof.* Part 2 follows from Newman's Lemma and Part 1. Proof of Part 1: " $\Rightarrow$ ": It follows directly from the assumption of local confluence. " $\Leftarrow$ ": Assume that all minimal  $I$ -corners are joinable, but the program is not locally confluent, i.e., there exists an  $I$ -corner  $\lambda'$  that is not joinable. From Lemma 5 we have that any  $\lambda'$  is subsumed by a minimal  $I$ -corner  $\lambda$  which by assumption is joinable. Part 3 is demonstrated by the following Example 11.  $\square$

In the general case, Theorem 6 does not provide an immediate recipe for proving local confluence due to the potentially infinite number of cases. The following example demonstrates two ways that this may appear.

**Example 11.** Consider a program that includes the following rules.

$r_1$ :  $p(X) \Leftarrow q(X)$ .  
 $r_2$ :  $p(X) \Leftarrow r(X)$ .  
 $r_3$ :  $p(X) \Leftarrow X \geq 1 \mid s(X)$ .

We assume the invariant

$$I(\langle S, T \rangle) \Leftrightarrow S \text{ is ground.}$$

There are no propagation rules, so we ignore the propagation history and consider a state as a multiset of constraints. Rules  $r_1$  and  $r_2$  give rise to a most general critical  $\alpha_1$ - $I$ -pre-corner  $(\circ \xleftarrow{R_1^x} \{p(X)\} \xrightarrow{R_2^x} \circ)$ . It has the following infinite set of minimal  $\alpha_1$ - $I$ -corners.

$$\{(\{q(t)\} \xleftarrow{R_1^t} \{p(t)\} \xrightarrow{R_2^t} \{q(t)\}) \mid t \text{ is a ground term}\}$$

Obviously, there is no least  $\alpha_1$ - $I$ -corner for this  $\alpha_1$ - $I$ -pre-corner. This problem was also noticed by Duck et al in their paper on observable confluence [15]. An additional consequence of our definitions is that a guard with an incomplete predicate may also give rise to an infinite set of minimal  $\alpha_1$ - $I$ -corners, even when we relax the invariant to equality. Now, rules  $r_1$  and  $r_3$  give rise to a most general critical  $\alpha_1$ - $I$ -pre-corner  $(\circ \xleftarrow{R_1^x} \{p(X)\} \xrightarrow{R_3^x} \circ)$ . It has the following infinite set of minimal  $\alpha_1$ - $I$ -corners.

$$\{(\{q(t)\} \xleftarrow{R_1^t} \{p(t)\} \xrightarrow{R_3^t} \{r(t)\}) \mid t \text{ is a ground term that can be read as a numeral } \geq 1\}$$

The solution that we describe in Section 5, and which has no counterpart in [15], is to consider each most general critical pre-corner (of which there are only finitely many) one at a time, lifted to a meta-level where we can reason about their joinability properties without having to expand them to a set of minimal  $I$ -corners.

A partial version of the classical results by Abdennadher et al [1, 3, 21] can be described as a special case of Theorem 6 with trivial invariant.

**Lemma 6 (Least I-corner Lemma; logical case with trivial invariant and  $\approx$ ).** Assume a program  $\Pi$  with logical and complete built-ins, invariant  $I(\cdot) \Leftrightarrow \text{true}$  and state equivalence relation  $=$ . The set of minimal  $\alpha_1$ -corners is finite and consists of least  $\alpha_1$ -corners, each of which is produced from an  $\alpha_1$ -pre-corner as follows:

- for each  $\alpha_1$ -pre-corner of the form  $(\circ \xleftarrow{R_1} \Sigma \xrightarrow{R_2} \circ)$  construct the unique  $\alpha_1$ -corner,  $(\Sigma_1 \xleftarrow{R_1} \Sigma \xrightarrow{R_2} \Sigma_2)$ .

*Proof.* Let us consider an  $\alpha_1$ -pre-corner  $\Lambda = (\circ \xleftarrow{R_1} \langle S, T \rangle \xrightarrow{R_2} \circ)$ . The indicated  $\alpha_1$ -corners do exist as the indicated derivation steps exist: by construction of  $\Lambda$ , the two application instances  $R_1, R_2$  exist (cf. Definition 8, p. 9: trivial guards and the condition of not modifying head constraints guaranteed); and they can apply to  $\langle S, T \rangle$  as their head constraints are in  $S$ , and no application record for  $R_1$  or  $R_2$  is in  $T$ . Referring to Proposition 1, p. 10 (functional dependency  $\text{Ancestor-state} \times \text{Application-instance} \rightarrow \text{Result-state}$ ), it is sufficient only to consider the common ancestor states of the involved (pre-) corners.

Let now  $\lambda$  be an  $\alpha_1$ -corner as stated in the lemma. First, we show that  $\lambda$  is minimal by contradiction, so assume the opposite, namely that there exists a  $\lambda \succ \lambda' > \Lambda$ ; let  $\langle s', t' \rangle$  refer to the common ancestor state of  $\lambda'$ . From  $\lambda' > \Lambda$  it follows that  $s' = s\theta' \uplus s'^+$  for some  $\theta', s'^+$ , and from  $\lambda \succ \lambda'$  that  $s = s'\theta \uplus s^+$  for some  $\theta, s^+$ ; thus  $s = s\theta\theta' \uplus s'^+\theta \uplus s^+$  and hence  $s'^+ = s^+ = \emptyset$  and  $\theta, \theta'$  are renaming substitutions.

In a similar way, we obtain

- $t' = t \uplus t'^+ \setminus t'^\div$  where any index of  $t'^+$  is in  $s'^+ = \emptyset$ , and thus  $t'^+ = \emptyset$ , and  $t'^\div \in t$  (cf. Definitions 13, 14); hence  $t' = t \setminus t'^\div$ ,
- $t = t' \uplus t^+ \setminus t^\div$  where any index of  $t^+$  is in  $s^+ = \emptyset$ , and thus  $t^+ = \emptyset$ , and  $t^\div \in t'$  as above; hence  $t = t' \setminus t^\div = t' \setminus t'^\div \setminus t^\div$ .

It follows now that  $t^\div = t'^\div = \emptyset$  and thus  $\lambda = \lambda'$ . Contradiction.

It remains to show that  $\lambda$  is a least corner for  $\Lambda$ , i.e., for any  $\lambda' > \Lambda$  it holds that  $\lambda' \succ \lambda$ . This follows from the fact that an unfolding of these two statements according to their respective definitions, inserting the same common ancestor state  $\langle s, t \rangle$  of  $\Lambda$  and  $\lambda$ , yields identical results.  $\square$

The most significant difference in the confluence results with the different semantics appears when guards contain incomplete built-ins. This implies cases where our semantics cannot apply, but the previous ones can, and thus local confluence is a stronger property with those semantics.

**Example 12.** Consider a program consisting of the following rules; invariant and equivalence are trivial and not considered.

```

r1: p(X) <=> 1 >= X, X >= -1 | q(X)
r2: p(X) <=> r(X)
r3: q(X) <=> 1 >= X, X >= 0 | r(x)
r4: q(X) <=> 0 >= X, X >= -1 | r(x)

```

As discussed in Section 3.3 and Example 7 above, the semantics of [1, 3, 15, 21] include a built-in store in the state, and a rule can fire when its guard is a consequence of the current built-in store. The built-in store for the common ancestor state of a critical pair is formed by the conjunction of the guards of the involved rules; ignoring global variables and propagation history, we obtain in the mentioned semantics the following critical pair, here shown with the ancestor state for ease of comparison.

$$\lambda^* = \left( \langle \{q(X)\}, (1 \geq X \wedge X \geq -1) \rangle \xrightarrow{r_1} \langle \{p(X)\}, (1 \geq X \wedge X \geq -1) \rangle \xrightarrow{r_2} \langle \{r(X)\}, (1 \geq X \wedge X \geq -1) \rangle \right)$$

This critical pair is not joinable as neither  $r_3$  nor  $r_4$  can apply to the left wing state since their respective guards are not consequences of the current built-in store. It follows that the program is not confluent when derivations are defined as by [1] and others.

With our semantics, the program is confluent. There are no corners similar to  $\lambda^*$  with  $p/1$  having an uninstantiated variable as its argument (the guard of  $r_1$  evaluates to *error* so  $r_1$  cannot apply). Instead we notice an infinite of family minimal corners for  $r_1$  and  $r_2$ , one for each numeral in the interval  $[-1, 1]$ ; for example:

$$\lambda^{0.5} = (\{q(0.5)\} \xleftarrow{R_1^{0.5}} \{p(0.5)\} \xrightarrow{R_2^{0.5}} \{r(0.5)\}) \quad \text{and} \quad \lambda^{-0.5} = (\{q(-0.5)\} \xleftarrow{R_1^{-0.5}} \{p(-0.5)\} \xrightarrow{R_2^{-0.5}} \{r(-0.5)\})$$

We see that  $\lambda^{0.5}$  can be extended to a joinability diagram by an application of  $r_3$  and the same for  $\lambda^{-0.5}$  by  $r_4$ . Obviously, the entire family of minimal corners is joinable, and the program is confluent under our semantics.

As mentioned, we do not intend to reason about infinite sets of minimal corners when it can be avoided. The methods introduced in the following section allows reasoning about abstract corners that visually resemble  $\lambda^*$  in Example 12 above, but in which the combined guard constraints are interpreted as meta-level restrictions on the intended instantiations of the states involved. (Such abstract corners are allowed to split, so our abstract version of  $\lambda^*$  in the example can split into two halves, one shown joinable using  $r_3$  and the other by  $r_4$ .)

## 5. Proving Confluence Modulo Equivalence using Abstract and Meta-level Constrained Corners and Diagrams

The classical approach to proving local confluence for CHR [1, 3] is distinguished by having to consider only a finite number of cases, each characterized by a critical pair of proper CHR states. Joinability of each critical pair is then shown by applying CHR rules directly.

As shown above, this does not generalize directly to the more general context with non-logical/incomplete built-in predicates and invariants. We introduce abstract states, that embed meta-level constraints derived from the invariant and rule guards, representing exactly the permissible states satisfying these constraints. Applications of CHR rules to abstract states are simulated with the meaning that they go only for these permissible states. This makes it possible to describe proofs of local confluence in terms of finitely many proof cases, also for examples where [15] requires infinitely many. Occasionally, we may need to split a case into sub-cases, each requiring different combinations of CHR rules for showing joinability.

Section 5.1 introduces the language METACHR, and Section 5.2 provides our central results on how confluence modulo equivalence may be shown by considering abstract corners constructed from the most general critical pre-corners.

### 5.1. A Meta-Language for CHR and its Semantics

In the following, we assume fixed sets of built-in and constraint predicates, invariant  $I$  and state equivalence  $\approx$ . The following definition introduces the basic elements of METACHR giving a parameterized representation for CHR and notions related to its semantics. Built-in predicates of CHR are lifted into METACHR in two ways, firstly by a lifted version of the *Exe* function (Section 3.1, p. 8) that is extended with an extra argument intended to hold the entire head of the actual rule instance, so the condition can be checked that a guard of a CHR rule cannot modify variables in that head. Secondly, each built-in predicate is represented as a predicate of the same name in METACHR expressing satisfiability of a given atom. The METACHR predicates *all-relevant-app-recs* and *common-vars* introduced below will be used to simulate details of CHR's derivation steps.

Two denotation functions will be defined, first  $\llbracket - \rrbracket^{Gr}$  that maps a ground METACHR term into a specific CHR related object, and next  $\llbracket - \rrbracket$  that maps a METACHR term parameterized by meta-variables into all the objects that it covers (analogous to subsumption above). Be aware that when a METACHR term is ground, it means that it contains no METACHR variables, although it may denote a non-ground CHR related object.

**Definition 17.** METACHR is a typed logical language; for given type  $\tau$ ,  $\text{META}_\tau$  ( $\text{META}_\tau^{Gr}$ ) refers to the set of (ground) terms of type  $\tau$ . The (ground) *denotation function* for each type  $\tau$  is a function

$$\llbracket - \rrbracket^{Gr} : \text{META}_\tau^{Gr} \rightarrow \text{CHR}_\tau$$

where  $\text{CHR}_\tau$  a suitable set of CHR related objects.

The types and terms of METACHR are assumed sufficiently rich such that any relevant object related to CHR is *denotable*, e.g., for any CHR state  $s$ , there exists a ground term  $t$  of METACHR with  $\llbracket t \rrbracket^{Gr} = s$ .

Whenever  $S, S_1, S_2$  are ground METACHR terms of type *state*, METACHR includes the following atomic formulas: invariant statements of the form  $I(S)$ , equivalence (statement)s of the form  $S_1 \approx S_2$ . We assume

similarly polymorphic operators for equality and various operations related to sets such as  $\in$ ,  $\subseteq$  etc. Each such predicate has a fixed meaning defined as follows; for any sequence of ground METACHR  $t_1, \dots, t_n$  of METACHR terms of relevant types,

$$p(t_1, \dots, t_n) \quad \text{if and only if} \quad p(\llbracket t_1 \rrbracket^{Gr}, \dots, \llbracket t_n \rrbracket^{Gr})$$

Whenever  $H$  is some ground term and  $G$  a ground term of type *guard*, the formula  $\widehat{Exe}(H, G)$  holds if and only if

- $Exe(\llbracket G \rrbracket^{Gr})$  is a proper substitution  $\theta$ ,
- $vars(\llbracket H \rrbracket^{Gr}) \cap domain(Exe\llbracket E \rrbracket^{Gr}) = \emptyset$ .

For each built-in predicate  $p/n$  of CHR, METACHR includes a *lifted* predicate  $p/n$  whose arguments are of type *term*, and which has a fixed meaning defined as follows; for ground terms  $T_1, \dots, T_n$ ,  $p(t_1, \dots, t_n)$  holds if and only if

- $Exe(p(\llbracket t_1 \rrbracket^{Gr}, \dots, \llbracket t_n \rrbracket^{Gr}))$  is a proper substitution.

Whenever  $t_1$  and  $t_2$  are ground terms, the predicate *common-vars*( $t_1, t_2$ ) holds if and only if

- $vars(\llbracket t_1 \rrbracket^{Gr}) \cap vars(\llbracket t_2 \rrbracket^{Gr}) \neq \emptyset$ .

METACHR includes a lifted version of the function *all-relevant-app-recs* (Def. 9) from terms of type *constraint-store* to sets of terms of type *application-record* defined such that *all-relevant-app-recs*( $s$ ) =  $t$  if and only if *all-relevant-app-recs*( $\llbracket s \rrbracket^{Gr}$ ) =  $\llbracket t \rrbracket^{Gr}$ .

For simplicity of notation, we assume for each predicate and function symbol, a function symbol in METACHR of similar arity and type *term* for its arguments, written with the same symbols. For example  $p(a, x)$  can be read as a ground METACHR term, and  $\llbracket p(a, x) \rrbracket^{Gr} = p(a, x)$  is a non-ground CHR term. To avoid ambiguity, METACHR variables are written by *italic* letters; this may occasionally clash the traditional use such letters for mathematical placeholders, and we add explanations when necessary to avoid confusion.

We extend the notational principle of indicating a state by one of its representations to METACHR as demonstrated in the following example.

**Example 13.** Assume a CHR constraint predicate  $p/2$ . The following equality between CHR states holds,

$$\llbracket \langle \{1:p(x, a)\}, \{r@1\} \rangle \rrbracket^{Gr} = \llbracket \langle \{2:p(y, a)\}, \{r@2\} \rangle \rrbracket^{Gr}$$

and thus the METACHR formula

$$\langle \{1:p(x, a)\}, \{r@1\} \rangle = \langle \{2:p(y, a)\}, \{r@2\} \rangle$$

is true.

The following notion of templates will be used for mapping specific CHR related objects, possibly containing variables, into a representation in METACHR with new METACHR variables, so that application of CHR substitutions are simulated by METACHR substitutions.

**Definition 18.** A *template*  $T'$  for a CHR related object  $t$  (e.g., term, constraint, rule, state, etc.) is a METACHR term formed as follows: 1) find a METACHR term  $T$  such that  $\llbracket T \rrbracket^{Gr} = t$ , and 2) form  $T'$  as a copy of  $T$  in which all subterms that are names of CHR variables are replaced systematically by new and unused METACHR variables.

For example, the METACHR term  $p(X, a)$  is a template for the CHR atom  $p(x, a)$ . Similar templates have been used in meta-interpreters for logic programs [10, 24], based on a lifting of the Prolog text into a meta-level representation in which Prolog unification is simulated by unification at the meta-level. The following definition is central. It is the basis for defining meta-level versions of derivations, corners and diagrams parameterized by METACHR variables that are constrained in suitable ways.

**Definition 19.** An *abstraction* of type  $\tau$  is a structure of the form

$$A_\tau \text{ WHERE } \Phi,$$

where  $A_\tau \in \text{META}_\tau$  and  $\Phi$  is a formula of METACHR referred to as a *meta-level constraint*. The abstraction

is *ground* if and only if  $A_\tau$  is ground and  $\Phi$  contains no free variables. In cases where the meta-level constraint is *true*, we may leave it out to simplify notation, i.e.,  $(A_\tau \text{ WHERE } \text{true})$  is written as  $A_\tau$ .

The denotation function  $\llbracket - \rrbracket^{Gr}$  is extended to ground abstractions and arbitrary structures (e.g., application instances, corners and diagrams) containing such, in the following way.

- For any ground abstraction  $A_\tau \text{ WHERE } \Phi$ ,

$$\llbracket A_\tau \text{ WHERE } \Phi \rrbracket^{Gr} = \begin{cases} \llbracket A \rrbracket^{Gr} & \text{whenever } \Phi \text{ is satisfied,} \\ \perp & \text{otherwise.} \end{cases}$$

- For any structure  $s(A_1, \dots, A_n)$  including ground abstraction  $A_1, \dots, A_n$ ,

$$\llbracket s(A_1, \dots, A_n) \rrbracket^{Gr} = \begin{cases} \perp & \text{if, for some } i, \llbracket A_i \rrbracket^{Gr} = \perp, \\ s(\llbracket A_1 \rrbracket^{Gr}, \dots, \llbracket A_n \rrbracket^{Gr}) & \text{otherwise.} \end{cases}$$

An abstraction or structure with abstractions  $\mathbf{A}$  is said to *cover* a concrete object or structure  $C$ , whenever there is a grounding METACHR substitution  $\sigma$  for which  $\llbracket \mathbf{A}\sigma \rrbracket^{Gr} = C \neq \perp$ . The set of all concrete objects or structures covered by  $\mathbf{A}$  is written  $\llbracket \mathbf{A} \rrbracket$ .

An abstraction or structure with abstractions  $\mathbf{A}$  is *consistent* whenever  $\llbracket \mathbf{A} \rrbracket \neq \perp$ .

Two abstractions or structures with abstractions,  $\mathbf{S}, \mathbf{S}'$  are *semantically equivalent* whenever, for any grounding substitution  $\sigma$  that  $\llbracket \mathbf{S}\sigma \rrbracket^{Gr} = \llbracket \mathbf{S}'\sigma \rrbracket^{Gr}$ . An abstraction of type state is referred to as an *abstract state*.

**Example 14 (Abstract States).** The abstract objects shown below include lifted versions of the CHR built-ins `constant/1` and `var/1` introduced in Definition 7 above. Notice in the lefthand sides that  $a, x$  are variables of METACHR.

$$\begin{aligned} \llbracket \langle \{p(a, x)\}, \emptyset \rangle \text{ WHERE } \widehat{Exe}(-, (\text{constant}(a), \text{var}(x))) \rrbracket &= \llbracket \langle \{p(a, x)\}, \emptyset \rangle \text{ WHERE } \text{constant}(a) \wedge \text{var}(x) \rrbracket \\ &= \{ \langle \{p(a, x)\}, \emptyset \rangle, \dots, \langle \{p(b, y)\}, \emptyset \rangle, \dots \} \\ &= \{ \langle \{p(a, x)\}, \emptyset \rangle \mid a \text{ is a constant, } x \text{ a variable} \} \\ \llbracket \langle \{p(a)\}, \emptyset \rangle \text{ WHERE } \text{var}(a) \wedge \text{const}(a) \rrbracket &= \emptyset \end{aligned}$$

In the example above, it was possible to turn a sequence of built-ins in a guard (the second argument of  $\widehat{Exe}(-, -)$ ) into a conjunction. However, this does not hold in general since different orders in a guard with non-logical or incomplete predicates may give different results.

Next, we introduce various building blocks, leading to abstract corners and joinability diagrams.

**Definition 20 (Abstract  $=$ ,  $\approx$  and  $I$ ).** Let  $T, T'$  be abstractions of the same type and  $S, S'$  abstract states. An *abstract equality* is a formula  $T = T'$ , an *abstract invariant* a formula  $I(S)$  and an *abstract equivalence* a formula  $S \approx S'$ . Let  $e(T_1, \dots, T_n)$ ,  $n = 1, 2$  be an arbitrary such formula;  $e(T_1, \dots, T_n)$  is defined to be true if and only if it the following properties hold.

- **(Soundness)** For any  $e(t_1, \dots, t_n) \in \llbracket e(T_1, \dots, T_n) \rrbracket$ , it holds that  $e(t_1, \dots, t_n)$  is true.
- **(Completeness)** For any  $i = 1, \dots, n$  and any  $t_i \in \llbracket T_i \rrbracket$  there exists a true atom  $e(t_1, \dots, t_n)$  with  $e(t_1, \dots, t_n) \in \llbracket e(T_1, \dots, T_n) \rrbracket$ .

An abstract state  $A$  for which  $I(A)$  holds is called an *abstract  $I$ -state*.

Notice that we do not require the constituents of abstract statements to be consistent, which means that two inconsistent abstract states will satisfy an abstract  $\approx$  statement. This is convenient for the formulation of the central Theorems 7 and 8, below.

The following property is useful when we want to build an abstract  $\beta$ -corner (defined below). For any abstract state, we can always produce an equivalence statement that covers all relevant equivalences at the level of CHR; this is made precise as follows.

**Proposition 4.** For any abstract state  $A$  there exist an abstract equivalence  $A \approx A'$  such that  $\llbracket A' \rrbracket = \{s' \mid s' \approx s \text{ for any } s \in \llbracket A \rrbracket\}$ .

*Proof.* Assume an abstract state of the form  $S \text{ WHERE } \Phi$ ,  $S$  some METACHR term of type *state* and  $\Phi$  a METACHR formula. The following abstract equivalence satisfies the proposition, where  $S'$  is a new and unused METACHR variable.

$$(S \text{ WHERE } \Phi) \approx (S' \text{ WHERE } S' \approx S \wedge \Phi).$$

□

As seen above, we have overloaded  $\approx$  to simplify notation. In the following example, we will elucidate the different levels of equivalence.

**Example 15 (Abstract Equivalence; Examples 1 and 2, continued).** We consider again the program that collects a set of items into a list with the suggested invariant and equivalence. For simplicity, we consider here only states containing a single **set** constraint whose argument is a list of constants. The propagation history is always empty, and we can ignore both that and the indices. In this example, and only here, we add subscripts to distinguish the different versions of the equivalence symbol  $\approx$ :  $\approx_{\text{CHR}}$ ,  $\approx_{\text{METACHR}}$ ,  $\approx_{\text{ABSTRACT}}$ . With these remarks, we can specify the equivalence at the level of CHR as follows:

$$\{\text{set}(\ell_1)\} \approx_{\text{CHR}} \{\text{set}(\ell_2)\} \Leftrightarrow \text{perm}(\ell_1, \ell_2),$$

where  $\text{perm}(\ell_1, \ell_2)$  is an auxiliary predicate that holds if and only if  $\ell_1$  and  $\ell_2$  are lists of constants that are permutations of each other. We will now demonstrate the construction given by the proof of Proposition 4 for an abstract state  $\{\text{set}([a, b])\} \text{ WHERE } \text{true}$  that we will write in the short form  $\{\text{set}([a, b])\}$ . The proposition suggests the following abstract equivalence that holds between the indicated abstract states.

$$\{\text{set}([a, b])\} \approx_{\text{ABSTRACT}} (S' \text{ WHERE } S' \approx_{\text{METACHR}} \{\text{set}([a, b])\})$$

To clarify the meaning of  $\approx_{\text{ABSTRACT}}$ , we unfold the right and innermost equivalence  $\approx_{\text{METACHR}}$  according to the definition, assuming a lifting of  $\text{perm}/2$  to METACHR, and we get the following.

$$\{\text{set}([a, b])\} \approx_{\text{ABSTRACT}} (\{\text{set}(\ell)\} \text{ WHERE } \text{perm}(\ell, [a, b]))$$

The right-hand side, is now in a form that makes it easier to apply abstract derivations.

With the abstract states at hand, we can now define abstract derivation steps.

**Definition 21 (Abstract Derivation Step).** An *abstract derivation step* is an abstraction of the form  $A \xrightarrow{D} A'$  where  $A$  and  $A'$  are abstract  $I$ -states,  $D$  an abstract label (i.e., abstract built-in atom or abstract application instance), with  $\text{vars}(A') \cup \text{vars}(D) \setminus \text{vars}(A)$  being fresh and unused variables, such that the following properties hold.

- **(Soundness)** For any  $(a \xrightarrow{d} a') \in \llbracket A \xrightarrow{D} A' \rrbracket$ , it holds that  $a \xrightarrow{d} a'$  is a concrete derivation step.
- **(Completeness)** For any  $a \in \llbracket A \rrbracket$  (for any  $a' \in \llbracket A' \rrbracket$ ) there exists a concrete derivation step  $(a \xrightarrow{d} a') \in \llbracket A \xrightarrow{D} A' \rrbracket$ .

*Abstract I-derivations* are defined in the usual way as a sequence of zero or more abstract  $I$ -derivation steps.

An abstract derivation step is intended to cover a set of concrete derivation steps, but unintended variable clashes in the abstract derivation step can cause undesired limitations on those. This is avoided with the requirement of fresh and unused variables. The second part of the completeness condition is relevant when we compose derivations and diagrams. If  $\llbracket A' \rrbracket$  includes an element  $a'$  for which the indicated step does not exist,  $A'$  is so to speak too big, and the next step (or equivalence statement) from  $A'$  would have to take care of too many irrelevant concrete states.

The following property follows immediately from the Definition 21.

**Proposition 5 (Abstract Derivation).** For any abstract  $I$ -derivation

$$\Xi = (A_0 \xrightarrow{D_1} A_1 \xrightarrow{D_2} \dots \xrightarrow{D_n} A_n) \quad , \quad n \geq 0.$$

the following properties hold.

- **(Soundness)** Any element of  $\llbracket \Xi \rrbracket$  is a concrete derivation.

- **(Completeness)** For any  $a_0 \in \llbracket A_0 \rrbracket$  (for any  $a_n \in \llbracket A_n \rrbracket$ ) there exists a concrete derivation  $(a_0 \xrightarrow{d_1} a_1 \xrightarrow{d_2} \dots \xrightarrow{d_n} a_n) \in \llbracket \Xi \rrbracket$ .

*Proof.* By induction. Base case  $n = 0$  is trivial; the step follows directly from Definition 21.  $\square$

Analogously to concrete corners, abstract corners are constructed using abstract derivations, abstract equivalence and abstract invariants, as follows.

**Definition 22 (Abstract Corners).** An *abstract I-corner* is a structure of the form  $(A_1 \text{ Rel}_1 A \text{ Rel}_2 A_2)$  where  $(A_1 \text{ Rel}_1 A)$  is an abstract  $I$ -derivation step or abstract equivalence, and  $(A \text{ Rel}_2 A_2)$  an abstract  $I$ -derivation step such that  $\text{vars}(A_1) \cap \text{vars}(A_2) \subseteq \text{vars}(A)$ .

We will refer to an abstract corner as an abstract  $\alpha_1$ -,  $\alpha_2$ -,  $\alpha_3$ -,  $\beta_1$ - or  $\beta_2$ -corner according to the relationships involved, analogous to what we have done for concrete corners (Definition 12, above). The following property is a consequence of what we have shown so far.

**Proposition 6.** For any abstract  $I$ -corner

$$\Lambda = (A_1 \text{ Rel}_1 A \text{ Rel}_2 A_2),$$

the following properties hold.

- **(Soundness)** Any element of  $\llbracket \Lambda \rrbracket$  is a concrete corner.
- **(Completeness)** For any  $a \in \llbracket A \rrbracket$  (for any  $a_1 \in \llbracket A_1 \rrbracket$ ) (for any  $a_2 \in \llbracket A_2 \rrbracket$ ) there exists a concrete corner  $(a_1 \text{ rel}_1 a \text{ rel}_2 a_2) \in \llbracket \Lambda \rrbracket$

*Proof.* The proposition is an immediate consequence of the soundness and completeness conditions in Definitions 20 and 21.  $\square$

Finally, we introduce abstract joinability diagrams for abstract corners, allowing us to treat a perhaps infinite set of corners in a single proof case.

**Definition 23.** An *abstract joinability diagram* (modulo  $\approx$ ) for an abstract  $I$ -corner is a structure of the form

$$\Lambda = (A_1 \text{ Rel}_1 A \text{ Rel}_2 A_2),$$

is a structure of the form

$$\Delta = (A'_1 \xleftarrow{*} A_1 \text{ Rel}_1 A \text{ Rel}_2 A_2 \xrightarrow{*} A'_2)$$

where  $A'_1 \xleftarrow{*} A_1$  and  $A_2 \xrightarrow{*} A'_2$  are abstract derivations such that the abstract equivalence  $A'_1 \approx A'_2$  holds. A given abstract corner is (*abstractly*) *joinable modulo  $\approx$*  whenever there exists an abstract joinability diagram for it.

**Proposition 7.** Let  $\Delta$  be an abstract joinability diagram for an abstract  $I$ -corner  $\Lambda$ . Then the following properties hold.

- **(Soundness)** Any element of  $\llbracket \Delta \rrbracket$  is a concrete joinability diagram.
- **(Completeness)** For any  $\lambda \in \llbracket \Lambda \rrbracket$  there exists a concrete joinability diagram for  $\lambda$  in  $\llbracket \Delta \rrbracket$ .

*Proof.* The proposition is an immediate consequence of soundness and completeness properties given by Propositions 5 and 6 and Definition 20.  $\square$

Combining this with the Critical Corner Theorem, Theorem 4, p. 15, we get immediately the following.

**Lemma 7 (Abstract Corner Lemma).** Assume a program  $\Pi$  with invariant  $I$  and state equivalence relation  $\approx$ , and let  $\mathcal{D}$  be a family of abstract  $I$ -corner that together covers all concrete corners that are subsumed by some general critical pre-corner for  $\Pi$ . Then  $\Pi$  is locally confluent modulo  $\approx$  if and only if all diagrams in  $\mathcal{D}$  is joinable.

In the following, we consider how to construct a family of abstract corners as required in Lemma 7.

## 5.2. Proving Confluence Modulo Equivalence using Abstract Joinability Diagrams

Here we will show how a set of most general critical pre-corners can be lifted to a set of abstract corners, and we identify necessary and sufficient conditions for confluence modulo equivalence. A program with invariant  $I$  and equivalence  $\approx$  is assumed.

A pre-corner (Definition 13, p. 13) is a common ancestor state whose wing states are indirectly characterized by their relationships to the ancestor state. Our way to lift it, to be defined below, consists of first lifting the common ancestor state, and then applying abstract versions of the indicated relationships (i.e., rule application, built-in or equivalence) to obtain abstract wing states, constrained at the meta-level by restrictions induced by guards and invariant.

As part of this, we need the following construction, which, for a given abstract ancestor state and type of derivation step, provides the resulting abstract state. For convenience, we combine derivation steps and  $\approx$ . We recall Proposition 1, p. 10, stating for the concrete case, that there is at most one resulting state for a derivation step.

**Definition 24.** Let  $A$  be an abstract  $I$ -state and  $Rel$  either an abstract derivation step or  $\approx$ . An *abstract post state for  $A$  with respect to  $Rel$*  is an abstract state  $A'$  such that  $A \text{ Rel } A'$  holds. Such a state  $A'$  is indicated as  $POST(A, Rel)$ .

Proposition 4, p. 21, shows that  $POST(A, \approx)$  can be found in a straightforward manner, although in practice it may be useful to unfold the definition of  $\approx$ .

For the definition to be useful for derivation steps, we assume that METACHR is sufficiently rich as to express an abstract state  $POST(A, Rel)$ . One way to obtain this is to include  $POST$  as a function in the language, whose meaning were defined semantically as indicated in Definition 24, but it will be more useful to define a procedure that produces an abstract state in terms of plain METACHR predicates and terms.

A general  $POST$  procedure that can handle all built-in predicates will be quite complex; Drabent's [13] analysis of a predicate transformer for unification of arbitrary terms demonstrates this. However, in many cases, the invariant and the selection state built-ins reduce the complexity. In all the examples we have considered, it has been straightforward to produce all necessary post states by hand; see, e.g., the larger example in Section 6.2 below.

**Example 16.** Let  $\Sigma$  be the abstract state  $(\langle \{p(x), x \text{ is } y\}, \emptyset \rangle \text{ WHERE } variable(x))$ , and we will construct a state  $POST(\Sigma, x \text{ is } y)$ .

This example is especially tricky as the built-in is incomplete and there are no restrictions on  $y$ , so the post state should capture both the *error* and proper states. We solve the problem, suggesting the following state; we assume two auxiliary METACHR predicates  $arithmetic(t)$  being true for any ground  $t$  for which  $\llbracket t \rrbracket^{Gr}$  can be evaluated as an arithmetic expression, and  $eval(t_1, t_2)$  being true for any ground  $t_1, t_2$  for which  $\llbracket t_1 \rrbracket^{Gr}$  can be evaluated as an arithmetic expression with value  $\llbracket t_2 \rrbracket^{Gr}$ .

$$\langle S, \emptyset \rangle \text{ WHERE } (S = \{p(y')\} \wedge eval(y, y') \wedge arithmetic(y)) \quad \vee \\ (S = \{error\} \wedge \neg arithmetic(y))$$

Here, the meta-variable  $x$  has been replaced by  $y'$ , which represents the value of the arithmetic expression  $y$ . Notice that no single rule can apply to this state, and if it happens to arise in an attempt to produce a joinability diagram, we need to apply the notion of splitting introduced below in Definition 26.

The following lifting of a pre-corner into an abstract corner is straightforward, although quite detailed as it includes meta-level versions of conditions for subsumption by pre-corner (Definition 14, p. 14). As we see in our examples, the detailed conditions often reduce to something much simpler, so the definition below represents, so to speak, worst cases.

**Definition 25 (Lifting Most General Critical Pre-Corners into Abstract Critical  $I$ -corners).**

An *abstract critical  $I$ -corner* for a most general critical pre-corner  $\Lambda = (\circ \text{ rel}_1 \langle s_0, t_0 \rangle \text{ rel}_2 \circ)$  is of the form

$$\mathbf{\Lambda} = (POST(A, Rel_1) \text{ Rel}_1 A \text{ Rel}_2 POST(A, Rel_2)),$$

where  $A$  is an abstract state, and  $Rel_1, Rel_2$  abstract derivation steps or  $\approx$ , specified as follows. Let firstly  $(\circ \text{ Rel}_1 \langle S_0, T_0 \rangle \text{ Rel}_2 \circ)$  be a template (Def. 18) for  $\Lambda$ . The construction of  $A$  depends on relationships  $Rel_1, Rel_2$ , that determine whether the corner is of type  $\alpha_1, \alpha_2$ , etc.



In case  $Rel_k$ ,  $k = 1, 2$  is an application instance, we assume the notation  $(r: H_k \leq G_k \mid C_k)$ . The symbols  $S^+, T^+, T^\dagger$  are fresh and unused variables, and let  $S$  stand for  $(S_0 \uplus S^+)$  and  $T$  for  $T_0 \uplus T^+ \setminus T^\dagger$ . (For the reading of the following, keep in mind that  $A, S, T, S_0, T_0, Rel_1, Rel_2, H_k, G_k, C_k$  are not METACHR variables, but mathematical placeholders. The predicates used below are elements of METACHR (Def. 17)) The common ancestor state  $A$  is given as follows for the different cases.

$$\begin{aligned}
\alpha_1: \quad & \langle S, T \rangle \text{ WHERE } I(\langle S, T \rangle) \wedge \widehat{Exe}(H_1, G_1) \wedge \widehat{Exe}(H_2, G_2) \wedge \\
& T^+ \subseteq \text{all-relevant-app-recs}(S_0 \uplus S^+) \setminus \text{all-relevant-app-recs}(S_0) \wedge \\
& T^\dagger \subseteq \text{all-relevant-app-recs}(S_0) \setminus \{\text{applied}(Rel_1), \text{applied}(Rel_2)\} \\
\\
\alpha_2: \quad & \langle S, T \rangle \text{ WHERE } I(\langle S, T \rangle) \wedge \widehat{Exe}(H_1, G_1) \wedge \\
& T^+ \subseteq \text{all-relevant-app-recs}(S_0 \uplus S^+) \setminus \text{all-relevant-app-recs}(S_0) \wedge \\
& T^\dagger \subseteq \text{all-relevant-app-recs}(S_0) \setminus \{\text{applied}(Rel_1)\} \wedge \\
& \text{common-vars}(G_1, Rel_2) \\
\\
\alpha_3, \beta_2: \quad & \langle S, T \rangle \text{ WHERE } I(\langle S, T \rangle) \wedge \\
& T^+ \subseteq \text{all-relevant-app-recs}(S_0 \uplus S^+) \setminus \text{all-relevant-app-recs}(S_0) \wedge \\
& T^\dagger \subseteq \text{all-relevant-app-recs}(S_0) \\
\\
\beta_1: \quad & \langle S, T \rangle \text{ WHERE } I(\langle S, T \rangle) \wedge \widehat{Exe}(H_1, G_1) \wedge \\
& T^+ \subseteq \text{all-relevant-app-recs}(S_0 \uplus S^+) \setminus \text{all-relevant-app-recs}(S_0) \wedge \\
& T^\dagger \subseteq \text{all-relevant-app-recs}(S_0) \setminus \{\text{applied}(Rel_1)\}
\end{aligned}$$

Whenever  $\Lambda$  is constructed as above, a semantically equivalent abstract corner  $\Lambda'$  may also be recognized as an abstract critical  $I$ -corner for  $\Lambda$ . By a *set of abstract critical corners for program  $\Pi$*  we mean a set consisting of one and only one abstract critical corner for each most general pre-corner for  $\Pi$ .

We notice that the set of all abstract critical  $I$ -corners for a program  $\Pi$  is finite as there exist only a finite number of most general critical pre-corners for  $\Pi$ , cf. Proposition 2.

**Proposition 8.** For any given program  $\Pi$  with invariant  $I$  and equivalence  $\approx$ , a set of abstract critical corners for it is finite.

**Lemma 8 (Cover by Abstract Critical Corner  $\Leftrightarrow$  Subsumed by Most Gen. Crit. Pre-Corner).**

For given program  $\Pi$ , let  $\Lambda$  be an abstract critical  $I$ -corner for a most general critical pre-corner  $\Lambda$ . Then the set of  $I$ -corners covered by  $\Lambda$  is identical to the set of  $I$ -corners subsumed by  $\Lambda$ . Furthermore,

$$\{\lambda \mid \exists \text{ abs. crit. corner } \Lambda \text{ for } \Pi . \lambda \in \llbracket \Lambda \rrbracket\} = \{\lambda \mid \exists \text{ most gen. critical corner } \Lambda \text{ for } \Pi . \Lambda < \lambda\}$$

*Proof.* The second part is a direct consequence of the first part.

For the first part, consider firstly an  $I$ -corner  $\lambda$  that is subsumed by a most general critical pre-corner  $\Lambda$  (with notation as in Definition 25 for each case of  $\alpha_1$ -,  $\alpha_2$ -, etc. corners); we prove that it is covered by the abstract critical corner  $\Lambda$  for  $\Lambda$  as given by the lemma as follows.

Subsumption means (by Definition 14, p. 14) that there exists a CHR substitution  $\theta$  and suitable sets  $s^+, t^+, t^\dagger$  and states  $post_1, post_2$  such that  $\lambda = (post_1 (rel_1 \theta) \langle s_0 \theta \uplus s^+, s_0 \uplus t^+ \setminus t^\dagger \rangle (rel_2 \theta) post_2)$  and the following conditions hold.

- $s = s_0 \theta \uplus s^+$
- $t = t_0 \uplus t^+ \setminus t^\dagger$
- $t^+ \subseteq \text{all-relevant-app-recs}(s_0 \theta \uplus s^+) \setminus \text{all-relevant-app-recs}(s_0 \theta)$
- $t^\dagger \subseteq \text{all-relevant-app-recs}(s_0 \theta)$

Let now  $\sigma$  be a METACHR substitution such that  $\llbracket S^+ \sigma \rrbracket^{Gr} = s^+$ ,  $\llbracket T^+ \sigma \rrbracket^{Gr} = t^+$  and  $\llbracket T^\dagger \sigma \rrbracket^{Gr} = t^\dagger$ . Since  $\langle S_0, T_0 \rangle$  is a template for  $\langle s_0, t_0 \rangle$ , and  $S^+, T^+, T^\dagger$  do not occur in  $\langle S_0, T_0 \rangle$ , we can extend  $\sigma$  such that  $\llbracket \langle S_0, T_0 \rangle \sigma \rrbracket^{Gr} = \langle s_0, t_0 \rangle$ . It remains, for each possible case of the corners being  $\alpha_1$ ,  $\alpha_2$ ,  $\alpha_3$ ,  $\beta_1$  or  $\beta_2$ , to

show that  $\Phi\sigma$  holds where  $A = (\langle S, T \rangle \text{ WHERE } \Phi)$  (i.e.,  $\Phi$  stands for the relevant of the alternative, detailed conditions in Definition 25, above). This can be verified by inspection in each case, referring to

1. the fact that  $\lambda$  is an  $I$ -corner, meaning that the two relationships  $(\text{post}_1 (\text{rel}_1\theta) \langle s_0\theta \uplus s^+, t_0 \uplus t^+ \setminus t^\div \rangle)$  and  $(\langle s_0\theta \uplus s^+, t_0 \uplus t^+ \setminus t^\div \rangle (\text{rel}_2\theta) \text{post}_2)$  do hold, i.e., their state arguments each satisfy the additional conditions, being the relevant of a derivation step (Definitions 8 and 10, p. 9-10) or an equivalence statement, and
2. the completeness parts of Definitions 20 and 21, and the relationship between  $\text{Exe}$  and  $\widehat{\text{Exe}}$  (Definition 17, p. 19).

The detailed arguments are left out as the  $\Phi$  formula in each case is a straightforward lifting of the similar conditions at the level of CHR.

The other way round, consider an  $I$ -corner  $\lambda = (a_1 \text{ rel}'_1 \langle s, t \rangle \text{ rel}'_2 a_2)$  covered by an abstract critical corner  $\Lambda$  (with notation as in Definition 25 for each case of  $\alpha_1$ -,  $\alpha_2$ -, etc. corners); we prove that it is subsumed by the most general critical pre-corner  $\Lambda$  given by the lemma (and notation as in Definition 25) as follows.

Covering means that there exists a grounding METACHR substitution  $\sigma$  such that  $\llbracket \langle S, T \rangle \sigma \rrbracket^{Gr} = \llbracket \langle S_0 \uplus S^+, T \uplus T^+ \setminus T^\div \rangle \sigma \rrbracket^{Gr} = \langle s, t \rangle$ , and, for  $k = 1, 2$ ,  $\llbracket \text{Rel}_k \rrbracket^{Gr} = \text{rel}'_k$  and  $\llbracket \text{POST}(\text{Rel}_k) \sigma \rrbracket^{Gr} = a_k$ , and the meta-level constraint part of  $A\sigma$  holds.

Now  $\langle S_0, T_0 \rangle$  is defined as a template for  $\langle s_0, t_0 \rangle$ , so there exists a METACHR substitution  $\sigma'$  such that  $\llbracket \langle S, T \rangle \sigma' \rrbracket^{Gr} = \langle s_0, t_0 \rangle$ , and thus we can find a CHR substitution  $\theta$  such that  $\langle s_0, t_0 \rangle \theta = \langle s, t \rangle$ .

Let now  $s^+ = \llbracket S^+ \sigma \rrbracket^{Gr}$ ,  $T_0 = \llbracket t_0 \sigma \rrbracket^{Gr}$ ,  $T^+ = \llbracket t^+ \sigma \rrbracket^{Gr}$  and  $T^\div = \llbracket t^\div \sigma \rrbracket^{Gr}$ ; it follows that  $s = s_0\theta \uplus s^+$ ,  $t = t_0 \uplus t^+ \setminus t^\div$  and, by definition of the METACHR version of *all-relevant-app-recs*, that  $t^+ \subseteq \text{all-relevant-app-recs}(s_0\theta \uplus s^+) \setminus \text{all-relevant-app-recs}(s_0)$  and  $t^\div \subseteq \text{all-relevant-app-recs}(s_0\theta)$ .

We have now that  $s = s_0\theta \uplus s^+$  and  $t = t_0 \uplus t^+ \setminus t^\div$ , and together with the soundness parts of Definitions 20 and 21, and the relationship between  $\text{Exe}$  and  $\widehat{\text{Exe}}$  (Definition 17, p. 19), it follows that  $\lambda$  is subsumed by  $\Lambda$ .  $\square$

We notice the following straight-forward property, indicating that we can use existing, automatic confluence checkers (e.g., [28]) to classify further abstract corners as “trivially joinable”, so only those abstract corners whose joinability critically depend on  $I$  and  $\approx$  need to be considered.

**Proposition 9.** Consider a program with invariant  $I$  and equivalence  $\approx$ , and with only logical and  $I$ -complete built-ins, and let  $\Lambda$  be an abstract critical  $\alpha_1$ - $I$ -corner lifted from a most general critical pre-corner  $\circ \xleftarrow{R_1} \Sigma \xrightarrow{R_2} \circ$ . If the concrete corner  $\Sigma_1 \xleftarrow{R_1} \Sigma \xrightarrow{R_2} \Sigma_2$  exists and is joinable modulo  $=$  with invariant *true*,  $\Lambda$  is joinable modulo  $\approx$  with invariant  $I$ .

It does not hold that a program is confluent modulo  $\approx$  if and only if all of its abstract critical pairs are joinable. This is demonstrated by the following example.

**Example 17 (Continuing Example 12, p. 18).** Consider the program of Example 12; its two first rules leads to the following abstract critical corner  $\Lambda^{r_1, r_2}$  (there are no propagation rules, so we leave out the propagation history).

$$(\{q(x)\} \text{ WHERE } 1 \geq x \wedge x \leq -1) \xleftarrow{r_1} (\{p(x)\} \text{ WHERE } 1 \geq x \wedge x \leq -1) \xrightarrow{r_2} (\{r(x)\} \text{ WHERE } 1 \geq x, x \leq -1)$$

We recall the two remaining rules of this program, that may perhaps apply to a state consisting of a single  $q$  atom.

$$\begin{aligned} r_3: & \quad q(X) \leq 1 \geq x, x \geq 0 \mid r(x) \\ r_4: & \quad q(X) \leq 0 \geq x, x \geq -1 \mid r(x) \end{aligned}$$

It appears that none of these rules can apply to the left wing state so  $\Lambda^{r_1, r_2}$  is not joinable, although any concrete corner covered by it is joinable.

This phenomenon which is induced by the presence of non-logical and incomplete predicates motivates the following.

**Definition 26 (Split-joinability).** Assume a set of METACHR formulas,  $\{\Phi_i \mid i \in \text{Inx}\}$ , for some finite or infinite index set  $\text{Inx}$ , such that

$$\Phi \Leftrightarrow \bigvee_{i \in \text{Inx}} \Phi_i.$$

A *splitting* of an abstract corner

$$A' \text{ Rel}_1 (\Sigma \text{ WHERE } \Phi_i) \text{ Rel}_2 A''$$

is the set of abstract corners

$$\{(POST((\Sigma \text{ WHERE } \Phi_i), \text{Rel}_1) \text{ Rel}_1 (\Sigma \text{ WHERE } \Phi_i) \text{ Rel}_2 POST((\Sigma \text{ WHERE } \Phi_i), \text{Rel}_2)) \mid i \in \text{Inx}\}.$$

An abstract corner is *split-joinable* modulo  $\approx$  whenever it has a splitting  $\{\Lambda_i \mid i \in \text{Inx}\}$  such that each  $\Lambda_i$  is either inconsistent or joinable modulo  $\approx$ .

The following property follows immediately from the definition.

**Proposition 10.** For any splitting of an abstract corner  $\Lambda$  into  $\{\Lambda_i \mid i \in \text{Inx}\}$ , it holds that

$$\llbracket \Lambda \rrbracket = \bigcup_{i \in \text{Inx}} \llbracket \Lambda_i \rrbracket.$$

**Example 18 (continuing Example 17).** The non-joinable abstract critical corner  $\Lambda^{r_1, r_2}$  is split-joinable using the disjunction  $(1 \geq x \wedge x \geq 0) \vee (0 \geq x \wedge x \geq -1)$ . Notice that neither  $\Lambda^{r_1, r_2}$  nor any member of its splitting covers a concrete corner of the form  $(\dots \leftarrow \{p(X)\} \mapsto \dots)$ , where  $X$  is a CHR variable.

We notice that the invariant of groundedness does not in itself make splitting necessary, see, e.g., the example in Section 6.1, below. In Section 6.3 below, we show an example of a program that needs an infinite splitting, but still we can use the results in the present section to show confluence.

**Theorem 7 (Abstract Critical Corner Theorem).** A CHR program with invariant  $I$  and equivalence  $\approx$  is locally confluent modulo  $\approx$  if and only if each of its abstract critical  $I$ -corners is either inconsistent, joinable modulo  $\approx$ , or split-joinable modulo  $\approx$ .

*Proof.* Follows immediately from Critical Corner Theorem, i.e., Theorem 4, p. 15, Lemma 8 and Proposition 10.  $\square$

Combining this result with Theorem 5, p. 15, we arrive at our following central result.

**Theorem 8.** A terminating program with invariant  $I$  and equivalence relation  $\approx$  is confluent if and only if each of its abstract critical  $I$ -corners is either inconsistent, joinable modulo  $\approx$ , or split-joinable modulo  $\approx$ .

## 6. Examples

We show three examples of confluence proofs. First, we give all details for the very simple but highly motivating example appearing in the Introduction of this paper. Next, we consider a more complex program, the Viterbi algorithm expressed in CHR, for which we formalize invariant and equivalence and give the proof of confluence modulo equivalence. This is a practically interesting algorithm, and the example also demonstrates that our framework can deal with nontrivial reasoning about the propagation history. Finally, we show an example that our method is robust for some cases where a countably infinite splitting is needed, Section 6.3.

Confluence modulo equivalence of a CHR version of the union-find algorithm [40], which has been used as a test case for aspects of confluence, is demonstrated informally by [12]. A detailed analysis and proof in terms of abstract critical corners is planned to appear in a future publication.

### 6.1. The Motivating One-line Program shown Confluent Modulo Equivalence

In the Introduction, we motivated confluence modulo equivalence for CHR by a program consisting of the following single rule.

$$\text{set}(L), \text{item}(A) \Leftrightarrow \text{set}([A|L]).$$

Here we formalize the invariant  $I$  and equivalence  $\approx$  hinted in Examples 1 and 2, p. 2-3, and give a proof of confluence modulo  $\approx$ .

- $I(\langle S, T \rangle)$  if and only if
  - $S = \{\text{set}(L)\} \uplus \text{Items}$  where  $L$  is a list of constants,  $\text{Items}$  is a set of  $\text{item}/1$  constraints whose arguments are constants,
  - $T = \emptyset$ .
- $\langle S, T \rangle \approx \langle S', T' \rangle$ , if and only if
  - $I(\langle S, T \rangle)$  and  $I(\langle S', T' \rangle)$ ,
  - $S = \text{set}(L) \uplus \text{Items}$  and  $S' = \text{set}(L') \uplus \text{Items}$  such that  $L$  and  $L'$  are permutations of each other.

We identify the following two most general critical pre-corners for the program. To give a complete picture, we have not abbreviated the application instances that label the derivation steps, as we do in most other examples.

$$\begin{aligned} \Lambda_1 &= \left( \circ \begin{array}{c} \text{set}(L), \text{item}(A) \Leftrightarrow \text{set}([A|L]) \\ \text{set}(L), \text{item}(B) \Leftrightarrow \text{set}([B|L]) \end{array} \langle \{\text{set}(L), \text{item}(A), \text{item}(B)\}, \emptyset \rangle \right) \\ \Lambda_2 &= \left( \circ \begin{array}{c} \text{set}(L_1), \text{item}(A) \Leftrightarrow \text{set}([A|L_1]) \\ \text{set}(L_2), \text{item}(A) \Leftrightarrow \text{set}([A|L_2]) \end{array} \langle \{\text{set}(L_1), \text{item}(A), \text{set}(L_2)\}, \emptyset \rangle \right) \end{aligned}$$

We lift  $\Lambda_1, \Lambda_2$  to the following abstract critical  $I$ -corners according to Definition 25, p. 24. Trivially satisfied meta-level constraints are removed.

$$\begin{aligned} \Lambda_1 &= \begin{array}{ccc} \langle \{\text{set}(\ell), \text{item}(a), \text{item}(b)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}(\ell), \text{item}(a), \text{item}(b)\} \uplus S, \emptyset \rangle) & & \\ \downarrow \text{set}(\ell), \text{item}(a) \Leftrightarrow \text{set}([a|\ell]) & \searrow \text{set}(\ell), \text{item}(b) \Leftrightarrow \text{set}([b|\ell]) & \\ \langle \{\text{set}([a|\ell]), \text{item}(b)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([a|\ell]), \text{item}(b)\} \uplus S, \emptyset \rangle) & & \langle \{\text{set}([b|\ell]), \text{item}(a)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([b|\ell]), \text{item}(a)\} \uplus S, \emptyset \rangle) \end{array} \\ \\ \Lambda_2 &= \begin{array}{ccc} \langle \{\text{set}(\ell_1), \text{set}(\ell_2), \text{item}(a)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}(\ell), \text{set}(\ell_2), \text{item}(a)\} \uplus S, \emptyset \rangle) & & \\ \downarrow \text{set}(\ell), \text{item}(a) \Leftrightarrow \text{set}([a|\ell]) & \searrow \text{set}(\ell_2), \text{item}(a) \Leftrightarrow \text{set}([a|\ell_2]) & \\ \langle \{\text{set}(\ell_2), \text{set}([a|\ell])\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}(\ell_2), \text{set}([a|\ell])\} \uplus S, \emptyset \rangle) & & \langle \{\text{set}(\ell), \text{set}([a|\ell_2])\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}(\ell), \text{set}([a|\ell_2])\} \uplus S, \emptyset \rangle) \end{array} \end{aligned}$$

The abstract corner  $\Lambda_2$  is inconsistent because  $I$  does not accept a constraint store with more than one  $\text{set}$  constraint, and  $\Lambda_1$  is shown joinable modulo  $\approx$  by the following abstract diagram  $\Delta_1$ .

$$\begin{aligned} \Delta_1 &= \begin{array}{ccc} \langle \{\text{set}(\ell), \text{item}(a), \text{item}(b)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}(\ell), \text{item}(a), \text{item}(b)\} \uplus S, \emptyset \rangle) & & \\ \downarrow \text{set}(\ell), \text{item}(a) \Leftrightarrow \text{set}([a|\ell]) & \searrow \text{set}(\ell), \text{item}(b) \Leftrightarrow \text{set}([b|\ell]) & \\ \langle \{\text{set}([a|\ell]), \text{item}(b)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([a|\ell]), \text{item}(b)\} \uplus S, \emptyset \rangle) & & \langle \{\text{set}([b|\ell]), \text{item}(a)\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([b|\ell]), \text{item}(a)\} \uplus S, \emptyset \rangle) \\ \downarrow \text{set}([a|\ell]), \text{item}(b) \Leftrightarrow \text{set}([b, a|\ell]) & & \downarrow \text{set}([b|\ell]), \text{item}(a) \Leftrightarrow \text{set}([a, b|\ell]) \\ \langle \{\text{set}([b, a|\ell])\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([b, a|\ell])\} \uplus S, \emptyset \rangle) & \approx \approx \approx \approx \approx & \langle \{\text{set}([a, b|\ell])\} \uplus S, \emptyset \rangle \text{ WHERE } I(\langle \{\text{set}([a, b|\ell])\} \uplus S, \emptyset \rangle) \end{array} \end{aligned}$$

The program is terminating since each derivation step reduces the number of  $\text{item}$  constraints by one, so by Theorem 8 it follows the program is confluent modulo  $\approx$ .

## 6.2. Confluence Modulo Equivalence of the Viterbi Algorithm

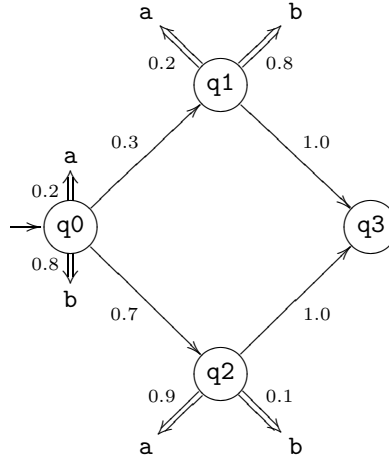
The Viterbi algorithm [41] is an example of a dynamic programming algorithm that searches for one optimal solution to a problem among, perhaps, several equally good ones.

A Hidden Markov Model, HMM, is a finite state machine with probabilistic state transitions and probabilistic emission of a letter from each state. The *decoding problem* for an observed sequence of emitted letters  $LS$  is that of finding a most probable *path* which is a sequence of state transitions that may have produced  $LS$ ; see [16] for a background on HMMs and their applications in computational biology.

A decoding problem is typically solved using the Viterbi algorithm [41] which is an example of a dynamic programming algorithm that produces solutions for a problem by successively extending solutions for growing subproblems. While there are potentially exponentially many different paths to compare, an early pruning strategy ensures linear time complexity.

The algorithm has been studied in CHR by [11, 12] as shown below. The optimal complexity requires a restriction in the possible derivations, namely that the **prune** rule (below) is applied as early as possible. In [11], it is demonstrated how such a rule ordering can be imposed by semantics-preserving program transformations; here we will show confluence of the program modulo a suitable equivalence (which ensures that limiting the rule order does not destroy the semantics of the program).

**Example 19.** The following diagram shows a very simple HMM with states  $q0, \dots, q3$ , emission alphabet  $\{a, b\}$  and probabilities indicated for transitions and emissions.<sup>12</sup>



The different events of transitions and emissions are assumed to be independent. For example, the sequence  $a \cdot b$  may be produced via the path  $q0 \cdot q1 \cdot q3$  with probability  $0.2 \cdot 0.3 \cdot 0.8 = 0.048$  or  $q0 \cdot q2 \cdot q3$  with probability  $0.2 \cdot 0.7 \cdot 0.1 = 0.014$ . For simplicity of the program that follows, it is assumed that an emission is produced when a state is left (rather than entered).

A specific HMM is encoded as a set of **trans/3** and **emit/3** constraints that are not changed during program execution.

**Example 20.** The HMM of Example 19 is encoded by the following constraints.

```

{ trans(q0,q1,0.3), trans(q0,q2,0.7), trans(q1,q3,1), trans(q2,q3,1),
  emit(q0,a,0.2), emit(q0,b,0.8),
  emit(q1,a,0.2), emit(q1,b,0.8),
  emit(q2,a,0.9), emit(q2,b,0.1)}

```

The CHR program that implements the Viterbi algorithm is as follows.

```

:- chr_constraint path/4, trans/3, emit/3.

```

<sup>12</sup> An interesting HMM will, of course, have loops so it can produce arbitrary long sequences. No explicit end states are needed.

```

expand @ trans(Q,Q1,PT), emit(Q,L,PE), path([L|Ls],Q,P,PathRev) ==>
  P1 is P*PT*PE | path(Ls,Q1,P1,[Q1|PathRev]).

```

```

prune @ path(Ls,Q,P1,_) \ path(Ls,Q,P2,_) <=> P1 >= P2 | true.

```

The meaning of a constraint  $\text{path}(Ls, q, p, R)$  is that  $Ls$  is a remaining emission sequence to be processed,  $q$  the current state of the HMM, and  $p$  the probability of a path  $R$  found for the already processed prefix of the emission sequence. To simplify the program, a path is represented in reverse order. The decoding of a sequence  $Ls$  starting from state  $q_0$  is stated by the query

:-  $HMM, \text{path}(Ls, q_0, 1, [q_0])$ .

where  $HMM$  is an encoding of a given HMM in terms of ground **trans** and **emit** constraints; for each pair of states  $q_1, q_2$ ,  $HMM$  contains at most one constraint of the form  $\text{trans}(q_1, q_2, \dots)$ , and for each pair of state  $q$  and emission letter  $L$ ,  $HMM$  contains at most one constraint of the form  $\text{emit}(q, L, \dots)$ .

The first rule of the program, **expand**, expands the existing paths and **prune** removes paths for identical subproblems (identified by the current HMM state and remaining emission sequence) with lower (or equal) probabilities. The program is terminating for such queries as any new **path** constraint introduced by the **expand** rule has a first argument shorter than that of its predecessor. A final state will include one **path** constraint of optimal probability for each prefix of the input string (unless the underlying state machine is not capable of generating that string).

The program is not confluent in the classical sense, as the **prune** rule may nondeterministically remove one or the other of two alternative **path** constraints of identical probability for the same sequence. In the following we introduce invariant  $I$  and equivalence  $\approx$  and show the program confluent modulo equivalence. For simplicity of the definitions and with no loss of generality, we assume a fixed indexed encoding  $HMM$  of a Hidden Markov Model with initial state  $q_0$  and fixed input emission sequence  $Ls_0$ .

**Definition 27.**  $I(\Sigma)$  if and only if  $\langle HMM \cup \{(0: \text{path}(Ls_0, q_0, 1, [q_0]))\} \rangle \xrightarrow{*} \Sigma$ .

However, in the proof of local confluence below, we will need a more direct characterization of the possible derivation states and the interrelations between their constraints. To this end, we state the following proposition.

**Proposition 11.** An  $I$ -state is of the form  $\langle S \cup HMM, T \rangle$  where  $S$  is a set of ground **path** constraints and  $T$  a propagation history.

For any  $(i: \text{path}([L|Ls], q, P, qs)) \in S$  for which  $\{(i^t: \text{trans}(q, q', P^t)), (i^e: \text{emit}(q, L, P^e))\} \in HMM$ , then one and only one of the following will be the case.

1. *Expansion has not taken place:*  
 $(\text{expand}@i^t, i^e, i) \notin T$
2. *Expansion produced and still in the store:*  
 $(\text{expand}@i^t, i^e, i) \in T \wedge \exists i'. (i': \text{path}(Ls, q', P', [q'|qs])) \in S$  where  $P'$  is the value of  $P * P^t * P^e$ .
3. *Expansion produced but pruned by stronger or equal alternative:*  
 $(\text{expand}@i^t, i^e, i) \in T \wedge \nexists i', P'. (i': \text{path}(Ls, q', P', [q'|qs])) \in S$   
 $\wedge \exists P', qs', i'. ((i': \text{path}(Ls, q', P', [q'|qs']))) \in S \wedge P' \geq P * P^t * P^e \wedge qs \neq qs'$

Notice in case 3, that the **path** required to exists may either be the stronger (or equal) alternative that via **prune** rule lead to the removal of  $\text{path}(Ls, q', P', [q'|qs])$ , or it may be an even stronger (or equal) one, meaning that several applications of **prune** have been involved. The uniqueness of **emit** (**trans**) constraints in  $HMM$  for a fixed  $q$  ensures that the constraints  $(i': \text{path}(Ls, q', P', [q'|qs]))$  in case 2 and 3 are unique and uniquely related to the application record  $\text{expand}@i_t, i_e, i$ .

*Proof.* We use induction over the length of the derivation leading to a given  $I$ -state.

*Base case.* The state  $\langle HMM \cup \{(0: \text{path}(Ls, q_0, 1, [q_0]))\} \rangle$  matches case 1 in the proposition.

*Step.* Assume an  $I$ -state  $\Sigma = \langle S \cup HMM, T \rangle$  satisfying the proposition, and let  $\Sigma \mapsto \Sigma^*$ , where  $\Sigma^* = \langle S^* \cup HMM, T \rangle$ . Two kinds of derivation steps are possible, one for each program rule.

**expand:** Assume that the **path** constraint  $i:\pi \in S$  of  $\Sigma$  is involved in an application of the **expand** rule. The only difference between  $\Sigma$  and  $\Sigma^*$  is that the latter includes a new **path** constraint  $i':\pi^* \in S^*$  and a

new application record  $(\text{expand}@i^t, i^e, i^*) \in T^*$ . In  $\Sigma^*$ ,  $i:\pi$  satisfies condition 2, and  $i^*:\pi^*$  condition 1; any other **expand** constraint in  $\Sigma^*$  satisfies the same of 1, 2, 3 as in  $\Sigma$ .

**prune:** Assume that the rule applies in  $\Sigma$  by the application instance  $i_1:\pi_1/i_2:\pi_2 \Leftrightarrow P_1 \triangleright P_2 \mid \text{true}$ . Thus  $S^* = S \setminus \{i_2:\pi_2\}$ ,  $T^* = T$ . It holds that, when  $i_1:\pi_1$  satisfies condition  $k$  in  $\Sigma$ , then it also satisfies condition  $k$  in  $\Sigma^*$  for  $k = 1, 2, 3$ .

The only way that the removal of  $i_2:\pi_2$  may affect the proposition is when there is another  $(i_2^0:\pi_2^0) \in S$  satisfying condition 2 or 3 with  $i_2$  in the role of the existentially quantified index  $i$  in either case. For condition 2,  $(i_2^0:\pi_2^0)$  satisfies condition 2 or 3 in  $\Sigma^*$  with  $i' = i_1$ ; for condition 3,  $(i_2^0:\pi_2^0)$  satisfies condition 3 in  $\Sigma^*$  with  $i' = i_1$ .  $\square$

Our equivalence relation specifies the intuition that two solutions for the same subproblem are equally good when they have the same probability. We recall that a state is defined as an equivalence class over state representations sharing the same pattern of variable recurrence.

**Definition 28.** The  $\approx$  is the smallest equivalence relation on  $I$ -states such that  $\langle S \cup HMM, T \rangle \approx \langle S' \cup HMM, T \rangle$  if and only if

- For any  $i:\text{path}(Ls, q, P, qs) \in S$ , there is an  $i:\text{path}(Ls, q, P, qs') \in S$ , and vice versa.

**Theorem 9.** The Viterbi program with invariant  $I$  is confluent modulo  $\approx$ .

*Proof.* According to Theorem 8, we can prove confluence of a CHR program by listing the set of critical abstract corners and showing each of them joinable or split joinable.

Firstly, we observe that no built-in predicate can appear in an  $I$ -state (they are only used in guards) and that the two built predicates  $\triangleright$  and **is** are  $I$ -complete. Thus, we have no  $\alpha_2$ - and  $\alpha_3$ -corners to consider, leaving only  $\alpha_1$ - and  $\beta$ -corners. For a better overview, we indicate the overall shapes of corners in the chosen canonical set, described in full detail below. There are three  $\alpha_1$ -corners, one for each possible way that two rules may produce a critical overlap:

$$\begin{aligned} \Lambda_1: & \quad \circ \xleftarrow{\text{prune}} \circ \xrightarrow{\text{prune}} \circ \\ \Lambda_2, \Lambda_3: & \quad \circ \xleftarrow{\text{prune}} \circ \xrightarrow{\text{expand}} \circ \text{ differing in whether or not the constraint being expanded is removed;} \\ & \quad \text{our analysis will show } \Lambda_3 \text{ (expanded constraint removed) is not joinable, but can be split into} \\ & \quad \text{three joinable subcases } \Lambda_3^{(1)}, \Lambda_3^{(2)}, \Lambda_3^{(3)}, \text{ one for each option in Proposition 11.} \end{aligned}$$

Two  $\beta$ -corners are found, one for each clause of the program.

$$\begin{aligned} \Lambda_4: & \quad \circ \approx \circ \xrightarrow{\text{prune}} \circ \\ \Lambda_5: & \quad \circ \approx \circ \xrightarrow{\text{expand}} \circ \end{aligned}$$

To save space, application steps are labelled by application records (rather than application instances) and we leave out also the  $id$  function, e.g., writing  $\text{prune}@_{\pi_1\pi_2}$  instead of  $\text{prune}@_{id((\pi_1, \pi_2))}$ .

We abbreviate the writing of the invariant in an abstract state, writing  $(\Sigma \text{ WHERE } I \wedge \dots)$  instead of  $(\Sigma \text{ WHERE } I(\Sigma) \wedge \dots)$ , where  $\Sigma$  is a (perhaps complex) abstract state expression. We use the following conventions in expressions that represent propagation histories.

- A condition of the form  $ra \notin T$ , where  $ra$  is a rule application and  $T$  a propagation history, may be removed in an abstract state expressions when it is clear from context that it is always satisfied. This is relevant when  $ra = (\langle \text{rule-id} \rangle @ \dots i \dots)$  and  $T$  is part of a state guaranteed not to contain  $i$ .
- When  $i$  represents a constraint index and  $T$  a propagation history, the notation  $T \setminus i$  is a shorthand for  $T \setminus \{ra \mid ra \text{ is an application record of the form } ra = (\langle \text{rule-id} \rangle @ \dots i \dots)\}$ .

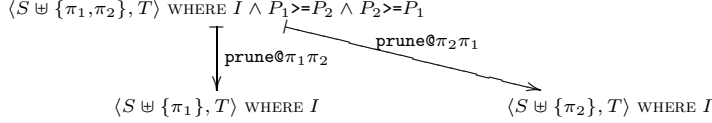
To simplify notation for the description of these corners, we introduce the following abbreviations; the recurrences of variables are significant.

$$\begin{aligned} \tau &= (i^t: \text{trans}(q, q', P^t)) \\ \eta &= (i^e: \text{emit}(q, L, P^e)) \\ \pi_j &= (i_j: \text{path}([L \mid LS], q, P_j, qs_j)) \quad \text{for } j = 1, \dots, 4 \\ \pi'_j &= (i'_j: \text{path}(LS, q', P'_j, [q' \mid qs_j])) \quad P'_j \text{ is the value of } P_j * P^t * P^e \text{ for } j = 1, \dots, 4 \end{aligned}$$

As it appears,  $\pi'_i$  can be derived from  $\pi_i$ ,  $\tau$  and  $\eta$  using the **expand** rule. The **path** constraints  $\pi_1, \dots, \pi_4$  all concern the same sub-problem, identified by the identical first and second argument,  $[L|LS]$  and  $q$ ; and analogously for the  $\pi'_i$  constraints.

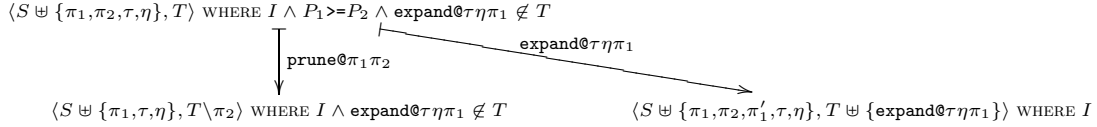
We consider now the canonical abstract corners one by one and show them (split) joinable.

$\Lambda_1$ : *Overlap of prune with itself*

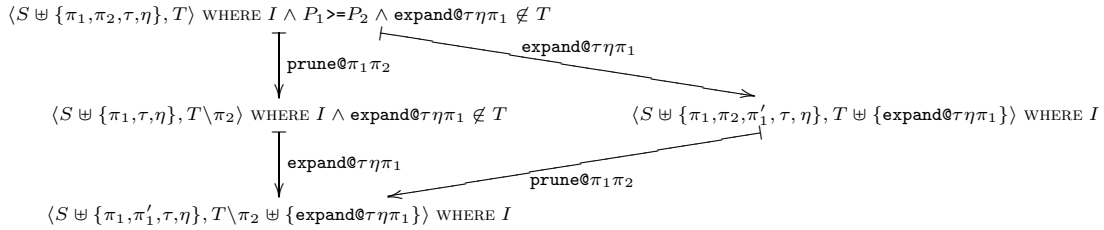


This extends immediately to a joinability diagram because the two abstract wing states are equivalent.

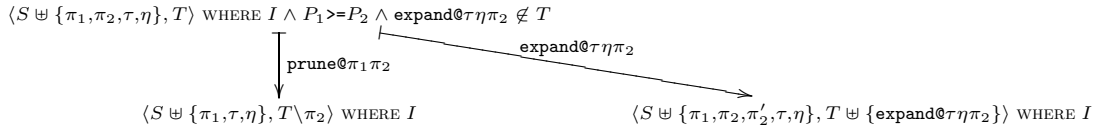
$\Lambda_2$ : *Overlap of prune and expand; expanded constraint not removed*



In this case, the two rules commute and the corner joins in one and the same abstract state.



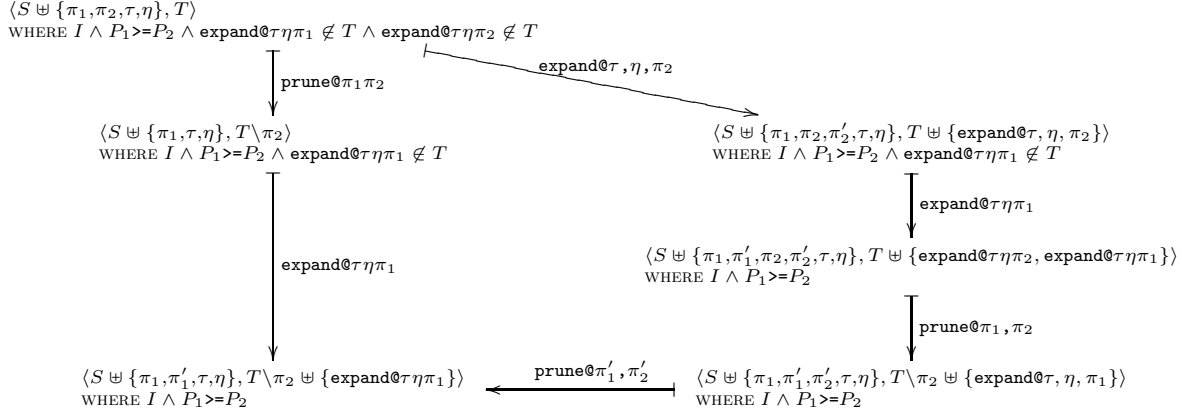
$\Lambda_3$ : *Overlap of prune and expand; expanded constraint removed*



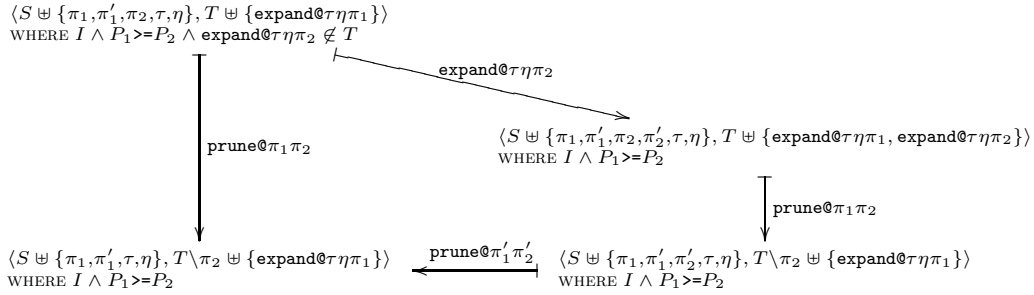
This abstract corner is not joinable as different derivations are possible depending on which of the three cases in Proposition 11 that holds for the path constraint  $\pi_1$ . This suggests a splitting of the corner into three new corners, that we can show joinable as follows. Hence, the corner is not joinable but split joinable. For reasons of space, we show only the related abstract joinability diagrams; the corners can be identified at the top.



$\Lambda_3^{(1)}$ : Split of  $\Lambda_3$ ;  $\pi_1$  applicable



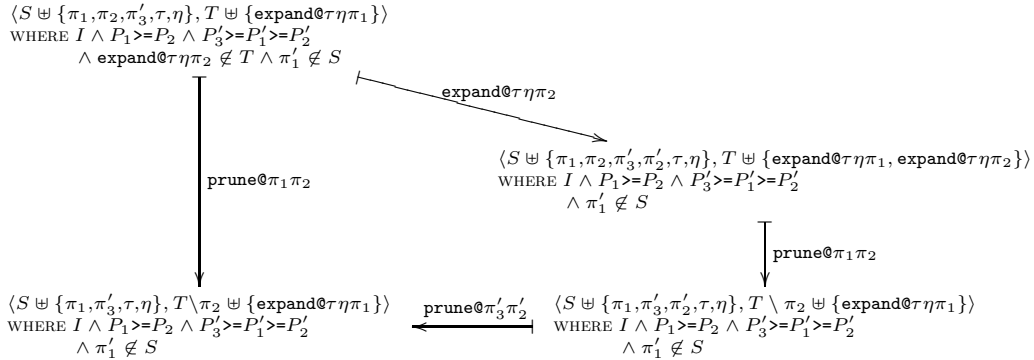
$\Lambda_3^{(2)}$ : Split of  $\Lambda_3$ ;  $\pi_1$  already expanded into  $\pi'_1$ ;  $\pi'_1$  still in state



Notice for the last  $\text{prune@}\pi'_1\pi'_2$  step, that the application history has no mentioning of  $\pi'_2$ , as the only event, since it was produced, is the step labelled  $\text{prune@}\pi_1\pi_2$ .

$\Lambda_3^{(3)}$ : Split of  $\Lambda_3$ ;  $\pi_1$  already expanded into  $\pi'_1$ ;  $\pi'_1$  already removed

As given by Proposition 11, option 3, this implies the presence in the common ancestor state of a **path** constraints  $\pi_3$ , with sufficiently high probability to have pruned  $\pi'_1$  as well as a possible  $\pi'_2$  (expanded from  $\pi'_2$  using  $\tau$  and  $\eta$ ). We can thus write this abstract corner and expand it to an abstract joinability diagram as follows.

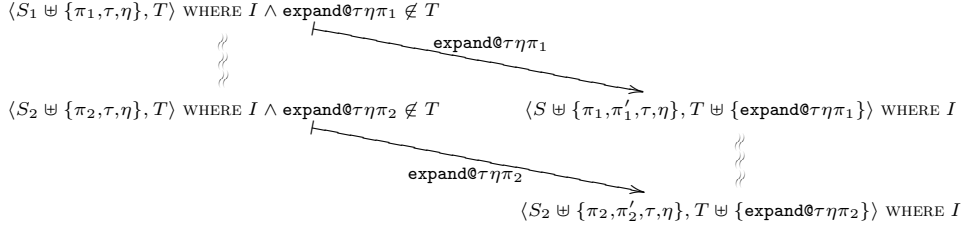


For the last  $\text{prune@}\pi'_3\pi'_2$  step, the application history has no mentioning of  $\pi'_2$ , as the only event since it was produced is the step labelled  $\text{prune@}\pi_1\pi_2$ .

This finishes the proof that  $\Lambda_3$  is split joinable. Now we turn to the canonical abstract  $\beta$ -corners of which there are two,  $\Lambda_4$  and  $\Lambda_5$ , one for each program rule.

#### $\Lambda_4$ : Equivalence and the **expand** rule

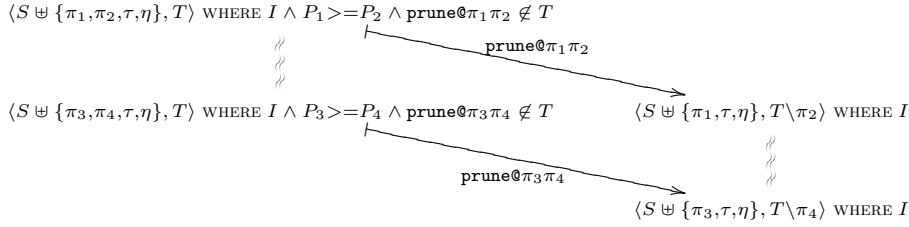
For the two equivalent states on the left side, it holds that  $\mathbf{expand@}\tau\eta\pi_2 = \mathbf{expand@}\tau\eta\pi_1$  and that  $S_i = HMM \uplus S'_i$ ,  $i = 1, 2$  where  $S'_1$  and  $S'_2$  consist of pairwise similar **path** constraints with identical index and that may differ only in their last arguments, and similarly for  $\pi_1$  and  $\pi_2$ .



To see that the lower equivalence holds, we notice that the indices of  $\pi'_1$  and  $\pi'_2$  can be chosen identical (and different from any other index used), and they may differ only in their last arguments.

#### $\Lambda_5$ : Equivalence and the **prune** rule

For the two equivalent states on the left side, it holds that  $\mathbf{expand@}\tau\eta\pi_{i+2} = \mathbf{expand@}\tau\eta\pi_i$ ,  $i = 1, 2$ , and that  $\pi_{i+2}$  and  $\pi_i$ ,  $i = 1, 2$ , may differ only in their last arguments. Furthermore,  $S_i = HMM \uplus S'_i$ ,  $i = 1, 2$  where  $S'_1$  and  $S'_2$  consist of pairwise similar **path** constraints with identical index and that may differ only in their last arguments.



Thus the set of abstract, critical corners have been shown joinable or split joinable; by termination and Theorem 8, the program is confluent modulo  $\approx$ .  $\square$

### 6.3. Countably Infinite Splitting

Here we show a program whose proof of confluence needs an infinite splitting of an abstract critical corner. The following CHR program is intended for queries of the form **start**,  $c(s^n(0))$ , where  $s^n(0)$  denotes the  $n$ th successor of 0 for any  $n \geq 0$ , e.g.,  $s^2(0) = s(s(0))$ .

```

easy    @    start      <=> easy.
hard    @    start      <=> hard.
done    @    c(X), easy  <=> c(0), end.
step    @    hard \ c(s(X)) <=> c(X).
finally @    c(0) \ hard <=> end.

```

The first step in such a derivation will introduce either an **easy** or a **hard** constraint. In case of **easy**, the derivation terminates after one additional step in the state  $\{c(0), \mathbf{end}\}$ . In case of **hard**, the derivation terminates after  $n + 1$  steps in the same state, so the program is confluent (modulo trivial  $\approx =$ ) under the invariant implied by the intended initial states.

We can specify the invariant as follows, using the unary meta-level predicate  $\mathit{succ}(N)$ , satisfied if and only if  $N$  of the form  $s^n(0)$  for an arbitrary natural number  $\geq 0$ .  $I((S, T))$  holds if and only if

- $S = \{R, c(N)\}$  where  $R \in \{\mathbf{start}, \mathbf{hard}, \mathbf{easy}, \mathbf{end}\}$  and  $\mathit{succ}(N)$ ,
- $T = \emptyset$ .

There exists only one consistent abstract critical  $I$ -corner  $\Lambda$ , and it is based on the overlap of the rules **easy**

and **hard**. Notice that the invariant has been unfolded, which is a semantics-preserving transformation.

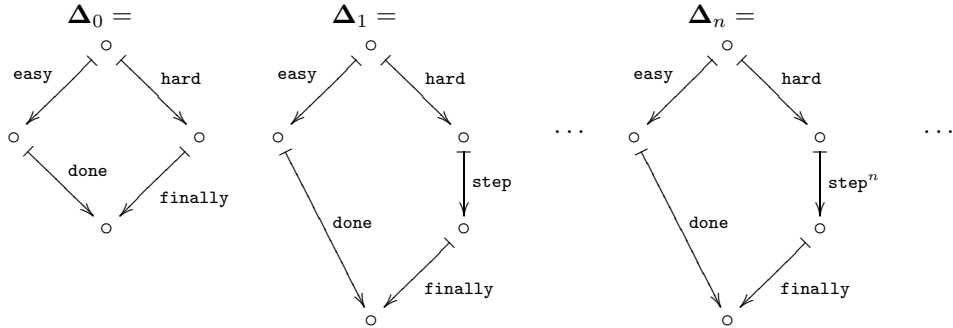
$$\Lambda = \begin{array}{ccc} \langle \{\mathbf{start}, c(n)\}, \emptyset \rangle \text{ WHERE } succ(n) & & \\ \text{easy@start} \downarrow & \swarrow \text{hard@start} & \\ \langle \{\mathbf{easy}, c(n)\}, \emptyset \rangle \text{ WHERE } succ(n) & & \langle \{\mathbf{hard}, c(n)\}, \emptyset \rangle \text{ WHERE } succ(n) \end{array}$$

The abstract critical corner  $\Lambda$  is not joinable as no single joinability diagram applies for all concrete corners covered by  $\Lambda$ . Therefore we split  $\Lambda$  using the infinite disjunction  $succ(n) \Leftrightarrow n = 0 \vee n = s(0) \vee \dots$ .

This leads to a countably infinite set of abstract corners  $\Lambda_0, \Lambda_1, \dots$ , where

$$\Lambda_i = \begin{array}{ccc} \langle \{\mathbf{start}, c(s^n(0))\}, \emptyset \rangle & & \\ \text{easy@start} \downarrow & \swarrow \text{hard@start} & \\ \langle \{\mathbf{easy}, c(s^n(0))\}, \emptyset \rangle & & \langle \{\mathbf{hard}, c(s^n(0))\}, \emptyset \rangle \end{array}$$

Each such abstract corner can be extended into a joinability diagram  $\Delta_i$ , each having  $i + 4$  abstract states and the same number of abstract derivation steps. For a better overview, we indicate only the shapes of these diagrams; the actual states are uniquely determined by the rules applied.



Thus  $\Lambda$  is split joinable, the program is terminating (no derivation starting from a state containing  $c(s^n(0))$  includes more than  $n + 2$  steps), and by Theorem 8 it follows that the program is confluent (modulo  $=$ ).

We notice here that confluence is due to the invariant; without invariant, we would get instead of  $\Lambda$ , the corner  $\langle \{\mathbf{easy}, c(x)\} \rangle \leftarrow \langle \{\mathbf{start}, c(x)\}, \emptyset \rangle \mapsto \langle \{\mathbf{hard}, c(x)\} \rangle$ . It is neither joinable nor split joinable.

## 7. Conclusions and Future Work

The aim of this paper is both theoretical and practical. Practical as it points forward to methods for proving highly useful properties of realistic CHR programs that may identify possible optimizations and contribute to correctness proofs; and theoretical since it provides a firm basis for understanding the notion of confluence modulo equivalence applied in the context of CHR.

We have demonstrated the relevance of confluence modulo equivalence for Constraint Handling Rules, which may also inspire to apply the concept to other systems with nondeterministic choice and parallelism. This may be approached either by migrating our results to other types of derivation systems, or using the fact that programs and systems of many such paradigms can be mapped directly into CHR programs; see an overview in the book by Frühwirth [22].

We introduced a new operational semantics for CHR that includes non-logical and incomplete built-ins and, as we have argued, this semantics is in many respects more in accordance with concrete implementations of CHR that what is seen in earlier work.

We introduced the idea of a logical meta-language METACHR specifically intended for reasoning about CHR programs, their semantics and their proofs of confluence modulo equivalence. These proofs are reified as collections of abstract joinability diagrams. A main advantage of this approach is that we can parameterize such proofs, i.e., diagrams, by meta-variables constrained at the meta-level to stand for, say, variables or

nonvariable terms of CHR. In our approach, this is essential for handling non-logical and incomplete built-ins correctly.

Our work is an improvement of the state-of-art in confluence proving for CHR [1, 3, 4, 15] in several ways: we generalize to modulo equivalence, we handle a larger and more realistic class of CHR programs, and for many programs we can reduce to a finite number of proof cases where [15] needs infinitely many, even for simple invariants such as groundness. The foundational works by Abdennadher et al [1, 3, 4] and Duck et al [15] use ordinary substitutions and inclusion of more constraints as their way to explain how their abstract cases, called critical pairs, cover large classes of concrete such pairs, each required to be joinable to ensure confluence. The use of the same language for abstract and concrete cases is quite limiting for what can be done at the abstract level, and which causes the mentioned problem of infinitely many proof cases. Taking the step that we do, introducing an explicit meta-language with meta-level constraints, eliminates this problem.

The use of a formal language provides a firm basis for automatic or semi-automatic support for deriving actual proofs, and our future plans include the development of such an implemented system. This requires a better understanding of how in general to construct abstract post states, given a state and an abstract derivation step; this is an important topic in our forthcoming research. It is obvious to incorporate an existing confluence checker in such a system in order to identify and eliminate those  $\alpha_1$  corners that are joinable even when invariant and equivalence are ignored.

One practical issue that needs to be understood better is how to cope with infinite splittings which have been exposed in our examples. It may be considered to allow meta-variables in METACHR to range over entire sub-derivations, suitably constrained at the meta-level. This may give rise to abstract diagrams that cover (in the formal sense we have defined) a range of differently shaped concrete diagrams. This potential is indicated informally in a diagram shown in Section 6.3, with a component indicating an entire sub-derivation, written as  $\xrightarrow{\text{step}^n}$ , so that we could show (still informally) an infinite set of corners joinable with a single argument.

A more detailed analysis of  $\beta$ -corners is desirable. We did not assume or impose any specific way of defining an equivalence, which means that any abstract  $\beta$ -corner needs to be considered as critical as soon as the equivalence is non-trivial. Huet [26] has shown a lemma for term rewriting systems, which will be interesting to adapt for CHR (see our Lemma 2, p. 5). It applies  $\approx = (\vdash)^*$  for some symmetric relation  $\vdash$ . Such a relation may be specified by a finite number of cases, as in a system of equations or logical equivalences. Here it seems possible to split each of our  $\beta$ -corners into a number of sub-cases, one for each case of the inductive definition of  $\vdash$ .

## References

- [1] Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: G. Smolka (ed.) CP, Constraint Programming, *Lecture Notes in Computer Science*, vol. 1330, pp. 252–266. Springer (1997)
- [2] Abdennadher, S., Frühwirth, T.W.: Integration and optimization of rule-based constraint solvers. In: M. Bruynooghe (ed.) Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25–27, 2003, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 3018, pp. 198–213. Springer (2003)
- [3] Abdennadher, S., Frühwirth, T.W., Meuss, H.: On confluence of Constraint Handling Rules. In: E.C. Freuder (ed.) CP, *Lecture Notes in Computer Science*, vol. 1118, pp. 1–15. Springer (1996)
- [4] Abdennadher, S., Frühwirth, T.W., Meuss, H.: Confluence and semantics of constraint simplification rules. *Constraints* 4(2), 133–165 (1999)
- [5] Aho, A.V., Sethi, R., Ullman, J.D.: Code optimization and finite Church-Rosser systems. In: R. Rustin (ed.) Design and Optimization of Compilers, pp. 89–106. Prentice-Hall (1972)
- [6] Aiken, A., Widom, J., Hellerstein, J.M.: Behavior of database production rules: Termination, confluence, and observable determinism. In: M. Stonebraker (ed.) Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992., pp. 59–68. ACM Press (1992)
- [7] Apt, K.R., Marchiori, E., Palamidessi, C.: A declarative approach for first-order built-in’s of Prolog. *Appl. Algebra Eng. Commun. Comput.* 5, 159–191 (1994)
- [8] Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1999)
- [9] Betz, H., Raiser, F., Frühwirth, T.W.: A complete and terminating execution model for constraint handling rules. *TPLP* 10(4–6), 597–610 (2010)
- [10] Christiansen, H.: CHR Grammars. *Int’l Journal on Theory and Practice of Logic Programming* 5(4–5), 467–501 (2005)
- [11] Christiansen, H., Have, C.T., Lassen, O.T., Petit, M.: The Viterbi algorithm expressed in Constraint Handling

- Rules. In: P. Van Weert, L. De Koninck (eds.) Proceedings of the 7th International Workshop on Constraint Handling Rules, Report CW 588, pp. 17–24. Katholieke Universiteit Leuven, Belgium (2010)
- [12] Christiansen, H., Kirkeby, M.H.: Confluence modulo equivalence in Constraint Handling Rules. In: M. Proietti, H. Seki (eds.) Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9–11, 2014. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 8981, pp. 41–58. Springer (2014)
- [13] Drabent, W.: A Floyd-Hoare method for Prolog. Tech. Rep. 2(13), Linköping Electronic Articles in Computer and Information Science (1997)
- [14] Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: B. Demoen, V. Lifschitz (eds.) Proc. Logic Programming, 20th International Conference, ICLP 2004, *Lecture Notes in Computer Science*, vol. 3132, pp. 90–104. Springer (2004)
- [15] Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In: V. Dahl, I. Niemelä (eds.) ICLP, *Lecture Notes in Computer Science*, vol. 4670, pp. 224–239. Springer (2007)
- [16] Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press (1999)
- [17] Falaschi, M., Gabbriellini, M., Marriott, K., Palamidessi, C.: Confluence in concurrent constraint programming. *Theor. Comput. Sci.* **183**(2), 281–315 (1997)
- [18] Frühwirth, T., Raiser, F. (eds.): Constraint Handling Rules, Compilation, Execution, and Analysis. Books on Demand GmbH, Norderstedt (2011)
- [19] Frühwirth, T.W.: User-defined constraint handling. In: D.S. Warren (ed.) Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, June 21–25, 1993, pp. 837–838. MIT Press (1993)
- [20] Frühwirth, T.W.: Constraint Handling Rules. In: A. Podelski (ed.) Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers, *Lecture Notes in Computer Science*, vol. 910, pp. 90–107. Springer (1994)
- [21] Frühwirth, T.W.: Theory and practice of Constraint Handling Rules. *Journal of Logic Programming* **37**(1–3), 95–138 (1998)
- [22] Frühwirth, T.W.: Constraint Handling Rules. Cambridge University Press (2009)
- [23] Haemmerlé, R.: Diagrammatic confluence for Constraint Handling Rules. *TPLP* **12**(4–5), 737–753 (2012)
- [24] Hill, P., Gallagher, J.: Meta-programming in logic programming. In: Handbook of Logic in Artificial Intelligence and Logic Programming, pp. 421–497. Oxford Science Publications, Oxford University Press (1994)
- [25] Holzbaur, C., Frühwirth, T.W.: A PROLOG Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence* **14**(4), 369–388 (2000)
- [26] Huet, G.P.: Confluent reductions: Abstract properties and applications to Knuth systems: Abstract properties and applications to Term Rewriting Systems. *J. ACM* **27**(4), 797–821 (1980)
- [27] Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: J. Leech (ed.) Computational Problems in Universal Algebras, pp. 263–297. Pergamon Press (1970)
- [28] Langbein, J., Raiser, F., Frühwirth, T.W.: A state equivalence and confluence checker for CHRs. In: P.V. Weert, L.D. Koninck (eds.) Proceedings of the 7th International Workshop on Constraint Handling Rules, Report CW 588, pp. 1–8. Katholieke Universiteit Leuven, Belgium (2010)
- [29] Mayr, R., Nipkow, T.: Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.* **192**(1), 3–29 (1998)
- [30] Newman, M.: On theories with a combinatorial definition of “equivalence”. *Annals of Mathematics* **43**(2), 223–243 (1942)
- [31] Niehren, J.: Uniform confluence in concurrent computation. *J. Funct. Program.* **10**(5), 453–499 (2000)
- [32] Niehren, J., Smolka, G.: A confluent relational calculus for higher-order programming with constraints. In: J. Jouannaud (ed.) Constraints in Computational Logics, First International Conference, CCL’94, Munich, Germany, September 7–9, 1994, *Lecture Notes in Computer Science*, vol. 845, pp. 89–104. Springer (1994)
- [33] Raiser, F., Betz, H., Frühwirth, T.W.: Equivalence of CHR states revisited. In: F. Raiser, J. Sneyers (eds.) Proc. 6th International Workshop on Constraint Handling Rules, Report CW 555, pp. 33–48. Katholieke Universiteit Leuven, Belgium (2009)
- [34] Raiser, F., Tacchella, P.: On confluence of non-terminating CHR programs. In: K. Djelloul, G.J. Duck, M. Sulzmann (eds.) Constraint Handling Rules, 4th Workshop, CHR 2007, pp. 63–76. Porto, Portugal (2007)
- [35] Personal communication with Tom Schrijvers (February 2016)
- [36] Schrijvers, T., Demoen, B.: The K.U.Leuven CHR System: Implementation and Application. In: T. Frühwirth, M. Meister (eds.) First Workshop on Constraint Handling Rules: Selected Contributions, pp. 1–5. Ulmer Informatik-Berichte, Nr. 2004-01 (2004)
- [37] Schrijvers, T., Frühwirth, T.W. (eds.): Constraint Handling Rules, Current Research Topics, *Lecture Notes in Computer Science*, vol. 5388. Springer (2008)
- [38] Sethi, R.: Testing for the Church-Rosser property. *J. ACM* **21**(4), 671–679 (1974)
- [39] Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules. *TPLP* **10**(1), 1–47 (2010)
- [40] Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. *J. ACM* **31**(2), 245–281 (1984)
- [41] Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* **13**, 260–269 (1967)

### A. Proof of Lemma 3: the Critical Corner Lemma

We recall the notation *all-relevant-app-recs*( $S$ ), Definition 9, p. 9, that refers to the set of all application records for rules of the current program taking indices from the constraint store  $S$ .

*Proof. (Lemma 3, p. 15)* We consider a program with invariant  $I$  and equivalence  $\approx$ , and we will go through the possible ways that an  $I$ -corner  $\lambda$  can be non-joinable and in each such case point out a most general pre-corner  $\Lambda$  that subsumes  $\lambda$ .

$\alpha_1$ :

Let  $\lambda = (\Sigma_1 \xleftarrow{R_1} \Sigma \xrightarrow{R_2} \Sigma_2)$  be an  $\alpha_1$ -corner that is not joinable with application instances  $R_k = (r_k : A_k \setminus B_k \leq g_k | C_k)$  for  $k = 1, 2$ . Let now  $H_k = A_k \cup B_k$ ,  $k = 1, 2$ , and  $Overlap = (B_1 \cap H_2) \cup (B_2 \cap H_1)$ .

In case  $Overlap = \emptyset$ , none of the application instances of  $\lambda$  remove any constraint from the common ancestor state that prevents the other one from being successively applied. Thus there exists some state  $\Sigma'$  such that  $\Sigma_1 \xrightarrow{R_2} \Sigma' \xleftarrow{R_1} \Sigma_2$  which means  $\lambda$  is joinable.

Assume now that  $Overlap \neq \emptyset$  and we proceed as follows to produce a most general  $\alpha$ -pre-corner  $\Lambda$  as follows. We select two most general application pre-instances

$$R_k^0 = (r_k : A_k^0 \setminus B_k^0 \leq g_k^0 | C_k^0), \quad i = 1, 2$$

in such a way such that, for  $k = 1, 2$ , the indices in  $R_k^0$  and  $R_k$  are pairwise identical, compared in the order they appear. Define also  $H_k^0 = A_k^0 \cup B_k^0$ ,  $k = 1, 2$ .

Let now, for  $k = 1, 2$ ,  $Overlap_k^0$  be the set of constraints in  $R_k^0$  whose indices coincide with those of  $Overlap$ . Since  $Overlap_1^0$  and  $Overlap_2^0$  have the common instance  $Overlap$ , there exists a most general unifier  $\sigma$  of  $Overlap_1^0$  and  $Overlap_2^0$ ; let furthermore  $\theta$  be a smallest substitution that such that  $Overlap_1^0 \sigma \theta = Overlap_2^0 \sigma \theta = Overlap$ .

Noticing that  $(g_1^0 \sigma, g_2^0 \sigma)$  is satisfiable (by  $\theta$ ), we can define now the following most general critical  $\alpha_1$ -pre-corner, that we argue below subsumes  $\lambda$ .

$$\begin{aligned} \Lambda^0 &= (\circ \xleftarrow{R_1^0 \sigma} \langle S^0, T^0 \rangle \xrightarrow{R_2^0 \sigma} \circ), \quad \text{where} \\ S^0 &= H_1^0 \sigma \cup H_2^0 \sigma \\ T^0 &= \text{all-relevant-app-recs}(H_1^0 \sigma \cup H_2^0 \sigma) \setminus \{r_1 @ id(A_1 B_1), r_2 @ id(A_2 B_2)\} \end{aligned}$$

We should check that, if  $r_1 = r_2$ , then  $A_1^0 \sigma \neq A_2^0 \sigma$  or  $B_1^0 \sigma \neq B_2^0 \sigma$  (cf. Def. 13) in order for the indicated  $\Lambda^0$  to actually be a most general  $\alpha_1$ -pre-corner: if this is not the case,  $\Sigma_1$  and  $\Sigma_2$  would be identical and thus  $\lambda$  joinable; contradiction.

Let now  $\Sigma = \langle s, t \rangle$  (i.e., we name the parts of the common ancestor in  $\lambda$ ) and we can define  $s^+$ ,  $t^+$  and  $t^\div$  as follows

$$\begin{aligned} s^+ &= s \setminus S^0 \theta \\ t^+ &= t \setminus T^0 \\ t^\div &= T^0 \setminus t \end{aligned}$$

By construction,  $R_k^0 \sigma \theta = R_k$ ,  $k = 1, 2$ , and we can show that the following properties hold.

$$\begin{aligned} s &= S^0 \theta \uplus s^+ \\ t &= T^0 \uplus t^+ \setminus t^\div \\ t^+ &\subseteq \text{all-relevant-app-recs}(S \theta \uplus s^+) \setminus \text{all-relevant-app-recs}(S \theta) \\ t^\div &\subseteq \text{all-relevant-app-recs}(S \theta) \end{aligned}$$

Thus, the conditions of Definition 14 are satisfied, proving that  $\Lambda^0$  subsumes  $\lambda$ .

$\alpha_2$ :

Let  $\lambda = (\Sigma_1 \xleftarrow{R} \Sigma \xrightarrow{b} \Sigma_2)$  be an  $\alpha_2$  corner that is not joinable, with application instance  $R = (r : A \setminus B \leq g | C)$ ,  $g$  non-logical or  $I$ -incomplete, and built-in  $b$ . Define now the following most general application pre-instance

$$R^0 = (r : H^0 \leq g^0 | C^0)$$

in such a way such that the indices in  $R^0$  and  $R$  are pairwise identical, compared in the order they appear, and let furthermore  $b^0$  be a most general indexed built-in atom with same predicate and index as  $b$ . We define now the following most general critical  $\alpha_2$ -pre-corner that we will argue subsumes  $\lambda$ .

$$\begin{aligned}\Lambda^0 &= (\circ \xleftarrow{R^0} \langle H^0, T^0 \rangle \xrightarrow{b^0} \circ), \quad \text{where} \\ T^0 &= \text{all-relevant-app-recs}(H^0) \setminus \{r@id(H^0)\}\end{aligned}$$

Let  $\theta$  be a smallest substitution such that  $R^0\theta = R$  and  $b^0\theta = b$ . If  $\text{vars}(b) \cap \text{vars}(g) = \emptyset$ , there would exists a state  $\Sigma'$  such that  $\Sigma_1 \xrightarrow{b} \Sigma' \xleftarrow{R} \Sigma_2$ ; contradiction. The remaining arguments to show that  $\Lambda^0$  subsumes  $\lambda$  are exactly as for the  $\alpha_1$  case.

$\alpha_3$ :

Let  $\lambda = (\Sigma_1 \xleftarrow{b_1} \Sigma \xrightarrow{b_2} \Sigma_2)$  be an  $\alpha_3$ -corner that is not joinable, with built-ins  $b_1, b_2$ , where  $b_1$  is non-logical or  $I$ -incomplete. We define now the following most general critical  $\alpha_2$ -pre-corner that we will argue subsumes  $\lambda$ .

$$\Lambda^0 = (\circ \xleftarrow{b_1^0} \langle H^0, \emptyset \rangle \xrightarrow{b_2^0} \circ)$$

To show that  $\Lambda^0$  subsumes  $\lambda$ , let  $\theta$  be a smallest substitution such that  $b_k^0\theta = b_k$ ,  $k = 1, 2$ , and proceed exactly as in the  $\alpha_1$  case (with  $T_0 = \emptyset$ ). We should also notice that it must hold that  $\text{vars}(b_1) \cap \text{vars}(b_2) \neq \emptyset$  as otherwise  $b_1$  and  $b_2$  would commute and  $\lambda$  be joinable.

$\beta_1$ :

Let  $\lambda = (\Sigma_1 \approx \Sigma \xrightarrow{R} \Sigma_2)$  be a  $\beta$ -corner that is not joinable, where  $R$  is an application instance with application instance  $R = (r: A \setminus B \leq g | C_k)$ . Define now the following most general application pre-instance

$$R^0 = (r: H^0 \leq g^0 | C^0)$$

in such a way such that the indices in  $R^0$  and  $R$  are pairwise identical, compared in the order they appear. The proof that the following most general critical  $\beta_1$ -pre-corner subsumes  $\lambda$  is similar to the previous cases.

$$\begin{aligned}\Lambda^0 &= (\circ \approx \langle H^0, T^0 \rangle \xrightarrow{R^0} \circ), \quad \text{where} \\ T^0 &= \text{all-relevant-app-recs}(H^0) \setminus \{r@id(H^0)\}\end{aligned}$$

$\beta_2$ :

Analogous to the  $\beta_1$  case and omitted.  $\square$